Titre: Title:	A fixed-point simd array processor and its applications to video compression coding
Auteur: Author:	Peijian Yuan
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Yuan, P. (2001). A fixed-point simd array processor and its applications to video compression coding [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8681/

UNIVERSITÉ DE MONTRÉAL

A FIXED-POINT SIMD ARRAY PROCESSOR AND ITS APPLICATIONS TO VIDEO COMPRESSION CODING

Peijian YUAN

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU GRADE DE MAÎTRIE ÈS SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

March 2001

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thése intitulée:

A FIXED-POINT SIMD ARRAY PROCESSOR AND ITS APPLICATIONS TO VIDEO COMPRESSION CODING

présentée par: Peijian YUAN

en vue de l'obtention du diplôme de: Maîtrise ès Sciences Appliquées

a été dûment accepté par jury d'examen constitué de:

M. SAWAN Mohamad,	Ph.D., président
M. BOIS Guy	Ph.D., membre et directeur de recherche
M. SAVARIA Yvon	Ph.D., membre et codirecteur de recherche
M. ABOULHAMID Mostapha	Ph.D., membre

Dedication

To my wife, Beisong

To my son, Max

Acknowledgements

It is my great pleasure to express sincere gratitude to these who have contributed to the successful completion of this work.

I have a deep feeling of indebtedness to my supervisor, Dr. Guy Bois for his constant confidence, support, encouragement, guidance and providing me financial support. His suggestions and criticisms were appreciated, and more importantly, his invaluable friendship will never be forgotten.

I am also thankful to my co-supervisor Dr. Yvon Savaria for introducing the thesis topic to me and providing me financial support to work in the PULSE research group.

I would like to express my appreciation to my colleagues Ivan Kralijic and Claude Villeneuve for helping me with using the PULSE simulation tools.

Many thanks to all my friends for helping to establish a convivial environment in which to work and for providing needed moral support during the difficult stages of this work.

Finally, I would like to thank my wife, Beisong Liu, to whom I owe much for her unending patience, encouragement and understanding, without which I could hardly have

completed this work. Special thanks are to my lovely son Max Yuan, for understanding me for not giving him full attention during his growing years.

Résumé

Cette thèse traite de l'utilisation de PULSE, un processeur à instruction unique et données multiples (SIMD) pour le Traitement et la Compression d'Images en format MPEG-2. De nos jours, les équipements de conférences vidéo, de téléphonie vidéo, de stockage d'images vidéo numérique, de télévision haute-définition (HDTV) et les systèmes de télévision et de multémedia numériques utilisent ce genre de fonctionnalité. Le stockage ou la transmission de données d'image numérique impliquent des bandes passantes et des quantités de mémoire importantes.

L'objectif principal de cette thèse est d'étudier les systèmes de codage de compression d'image. Elle traite notamment de la conception de systèmes de haute performance et elle étudie les compromis entre la précision et la complexité afin de réaliser des systèmes efficaces.

1) Le mémoire propose un algorithme précis et efficace pour effectuer la détection du mouvement dans un flot d'images. La méthode de recherche complète est rapide et précise. Elle est cependant très coûteuse. Une méthode de recherche graduelle mais complète permet de trouver le meilleur appariement avec un effort moyen réduit pour des images simples.

2) Une architecture adaptée à l'algorithme proposé est analysée et sa réalisation est décrite. Nos résultats démontrent qu'une puce PULSE permet de réaliser des systèmes de compression d'images efficaces et flexibles, qui exploitent un haut degré de parallélisme. Combiné avec un processeur de traitement de signal commercialement disponible, le C40 de la société Texas Instruments, on peut réaliser efficacement des systèmes de compression d'images de haute performance. Une telle architecture hétérogène est efficace et flexible.
3) Nous proposons aussi une méthode efficace pour le calcul de la transformée cosinus (DCT et IDCT) avec une puce PULSE. Cette méthode exploite une table de cosinus chargée dans les mémoires internes de PULSE pour éviter des calculs qui exigent un grand nombre d'opérations.

Des développements additionnels permettraient d'optimiser encore plus les algorithmes proposés afin d'accélérer la compression d'image avec PULSE.

Abstract

This thesis is concerned with applying a Fixed-point SIMD Array Processor PULSE to Image Compression with the MPEG-2 Standard. Video compressor is widely used in today's video conferences, videophone, digital video storage. Storage or transmission of digital images requires large memories and transmission bandwidth. This motivated research on this topic.

The main objective of this thesis is to study image compression coding systems. Several aspects of design for high-speed and high accuracy processing are considered. In order to realize a simple and effective image compression coding system, the following areas are investigated.

1) A high-speed and high accuracy algorithm for motion estimation is developed.
   The Gradual Full search method (GFSM) algorithm reduces the time required to find matches and no possible solution is neglected in the search area. Although the program is slightly more complex than the Full Searching Method (FSM), it is three times faster than FSM when processing a simple image.

2) Different system architectures for image compression coding are discussed and designed.

The results obtained during the research conducted for this thesis will prove that a PULSE chip can be used to construct flexible multi DSP systems, to accelerate image compression. Using PULSE chips with a C40 DSP and a FPGA control unit, we can construct a hardware/software system for image compression. It will not only reduce the cost of an image compression coding systems, but also improve its flexibility.

3) A simple and effective method to calculate DCT or IDCT with cosine functions using the PULSE chip is developed.

   A possible method to compute the cosine function uses exponential function. Calculating cosine function is relatively expensive. Thus we propose using precomputed tables stored in PULSE's internal memory to accelerate computation of DCT or IDCT.

   Further developments could improve the throughput of image compression on the PULSE chip.

Table of Contents

List of Figures

List of Tables

List of Appendix 85

CHAPTER 1

INTRODUCTION

A General Presentation of the Problem

This thesis presents a Hardware Software Co-design with the PULSE(Parallel Ultra Large Scale Engine) chip used for image processing. It reports or research carried as part of the PULSE project that led to the development of the PULSE chip. Nowadays, moving image coding systems have a very promising application field: Videoconferencing, Videophone, Digital Video Storage, High-Definition Television (HDTV), Digital Television and Multimedia Systems. In moving image coding systems, data compression is needed for efficient management of large amounts of information. For example, a color image with a resolution of 1000 by 1000 pixels (picture elements) occupies 3 megabytes of storage in an uncompressed form. Data compression is especially useful for the transmission of such high data through transmission channels. For instance, bit-rate ranges from 10 Mb/s for broadcast-quality video to more than 100 Mb/s for HDTV signals.

In order to reduce the transmission rate, using prediction techniques based on motion estimation. This scheme increases the compression ratio to 50~200:1. Motion Estimation is the most demanding part in the coding algorithm. For example, in an image coding system in MPEG2 standard (Figure 2-3), the computational power required is approximately 1.2 GOPS; and around 50% of this effort is devoted to motion estimation.

At the decoder, motion estimation is not necessary, therefore lower computational power is required.

Main Objective and Methodology

The main objective of this thesis is to study image compression coding systems and some popular algorithms used for that purpose. Several aspects of design for high-speed and high accuracy processing are considered. In order to realize a simple and effective image compression coding system, the PULSE chip is considered as a potential platform.

The PULSE chip is a new ultra-high performance SIMD (Single Instruction Multiple Data) architecture DSP (Digital Signal Processing) for high-end video and related applications. It has one controller and four process elements with clock of 54MHZ and 4 ports, having an I/O capability as high as 216 Mega words/sec. Its I/O ports are designed to allow forming linearly connected chains of chips. With this system architecture and using PULSE assembly language, a real time image processing system can be implemented.

Originality

1) A high-speed and high accuracy algorithm for motion estimation is developed.

   The Gradual Full search method (GFSM) algorithm reduces the time required to find matches and no possible solution is neglected in the search area. Although the

program is slightly more complex than the Full Searching Method (FSM), it is three times faster than FSM when processing a simple image.

2) A simple and effective method to calculate DCT or IDCT with cosine functions using the PULSE chip is developed.

   A possible method to compute the cosine function uses the exponential function. Calculating cosine function is relatively expensive. Thus we propose using pre-compute tables, stored in PULSE's internal memories, to accelerate computation of DCT or IDCT.

   Further developments could improve the throughput of image compression on the PULSE chip.

3) Different system architectures for image compression coding are discussed and designed.

The results obtained during the research conducted for this thesis will prove that a PULSE chip can be used to construct flexible multi DSP systems, and to accelerate image compression. Using PULSE chips with a C40 DSP and a FPGA control unit, we can construct a hardware/software system for image compression. It will not only reduce the cost of image compression coding systems, but also improve their flexibility.

Organization of the Thesis

Chapter 2 will introduce the MPEG2 standard, the PULSE chip, and review some previous research work. I will also describe some proposed algorithms. Chapter 3 describes the implementation of the convolution algorithms on PULSE. Chapter 4, 5 and 6 are the main parts of this thesis. Chapter 4 and 5 include the processing of motion estimation and DCT algorithms using the PULSE chip. A hardware and software co-design system using a C40 chip & PULSE chips is discussed in chapter 6. Chapter 7 summarizes our conclusions.

CHAPTER 2

A REVIEW OF IMAGE COMPRESSING ALGORITHMS AND THEIR PROCESSOR ARCHITECTURES

2.1 MPEG Standard

In today's world, videoconferencing, videophone, digital video storage, high-definition television (HDTV), digital television and multimedia systems are widespread. Storage or transmission of these data requires large memories and high bit-rate. Therefore, data compression has been a subject of intensive research and development for the past few years.

2.1.1 Background

MPEG is a video compression technology formulated by the Moving Pictures Experts Group, a joint committee of the International Standardization Organization (ISO). The first MPEG standard, known as MPEG-1, was formalized by the MPEG committee in January 1992.

MPEG-1 compression incorporates both audio and video. For NTSC video (United States and Japan) MPEG-1 uses the Standard Image Format (SIF) of 352x240 at 30 frames per second. Audio is 16-bit, stereo sampled at 44KHz. MPEG data rates are variable,

although MPEG-1 was designed to provide VHS video quality, and CD-ROM audio quality at a combined data rate of 1.2 megabits per second.

By resolution and data rate, MPEG-1 is targeted primarily at the computer and games markets. By contrast, MPEG-2, adopted in the spring of 1994, is a broadcast standard specifying 720x480 pixels resolution, playback at 60 fields per second and data rates ranging from two to 10 megabits per second. MPEG-2 is the core compression technology for DVD, the high-density CD-ROM standard that many fell will replace VHS tapes as the standard for consumer video.

MPEG-3 was dropped, and MPEG-4 is a very low-bit-rate codec targeting videoconferencing, Internet, and other low-bandwidth applications.

2.1.2 A Brief Overview of MPEG-2

1) What is MPEG-2

MPEG-2 is an audio/video compression/decompression standard. The audio/video inputs are compressed by an encoder, and decompressed by a decoder for playback.

The MPEG-2 standard is actually composed of three standards formulated by the Moving Pictures Experts Group, a working group of the International Organization for Standardization (ISO). ISO standard 13818-1 covers the MPEG-2 system stream, ISO standard 13818-2 addresses MPEG-2 video, and ISO standard 13818-3 describes MPEG-

2 audio. Work on MPEG-2 started back in 1988, and all three standards were finally approved in November 1994.

MPEG-2 video resolution can range from 720x480 to 1280x720, with the latter targeting high-definition television (HDTV) applications (cable 15.1). The most common resolution is 720x480, roughly the resolution of a full-screen NTSC (National Television Standards Committee) image. This contrasts with MPEG-1's maximum resolution of 352x240, or quarter-screen TV. While MPEG-1 is limited to 30 frames per second, MPEG-2 can operate at 60 fields, the scan rate of NTSC television, enhancing suitability for broadcast applications like HDTV, cable television, and broadcast satellite.

2) Video Compression Technology

Since MPEG-2 includes both audio and video, all MPEG-2 codecs must address both formats. The block diagram of an MPEG-2 encoder system is shown in Figure2-1.

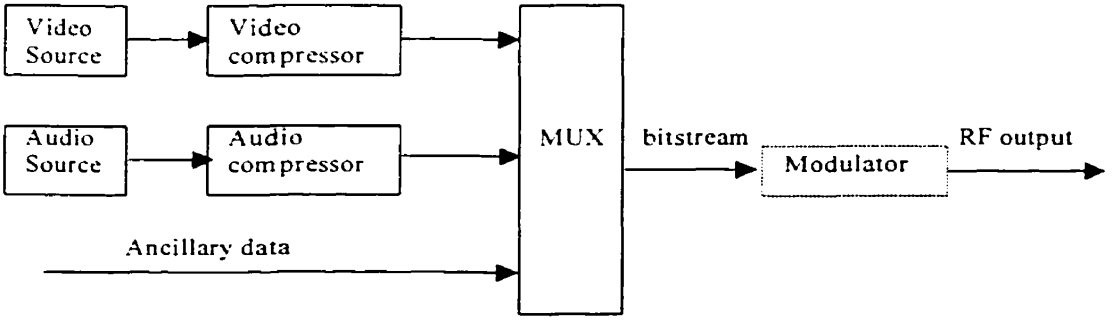


Figure2-1 The block diagram of MPEG-2 encoder system

This thesis is mainly focused on the implementation of the video compressor. MPEG video is specifically used in compression of video sequences which are simply a series of

pictures taken at closely spaced intervals in time. Except for the special case of a scene change, these pictures tend to be quite similar from one to the next. Intuitively, a compression system ought to be able to take advantage of this similarity.

The compression techniques (compression models) with MPEG take advantage of this similarity or predictability from one picture to the next in a sequence. Compression techniques that use information from other pictures in the sequence are usually called interframe techniques.

When a scene change occurs, interframe compression does not work and the compression model should be changed. In this case the compression model should be structured to take advantage of the similarity of a given region of a picture to immediately adjacent area in the same picture. Compression techniques that only use information from a single picture are usually called intraframe techniques. These two compression techniques, interframe and intraframe, are at the heart of the MPEG video compression algorithm.
Each video sequence is divided into one or more groups of pictures, and each group of pictures is composed of one or more pictures of three different types, I, P, and B, as illustrated in Figure 2-2. I-pictures (intra-coded pictures) are coded independently, entirely without reference to other pictures. P and B-pictures are compressed by coding the differences between the picture and reference I or P-pictures, thereby exploiting the similarities from one picture to the next.

P-pictures (predictive-coded pictures) obtain predictions from temporally preceding I or P-pictures in the sequence, whereas B-pictures (bi-directionally predictive-coded pictures) obtain predictions from the nearest preceding and / or upcoming I or P-pictures in the sequence. Different regions of B-pictures may use different predictions, and may predict from preceding pictures, upcoming pictures, both, or neither. Similarly, P-pictures may also predict from preceding pictures or use no prediction. If no prediction is used, that region of the picture is coded by intraframe techniques.

In a closed group of pictures, P and B-pictures are predicted only from other pictures in that group of pictures; in an open group of pictures, the prediction may be from pictures outside of the group of pictures [MPG97].

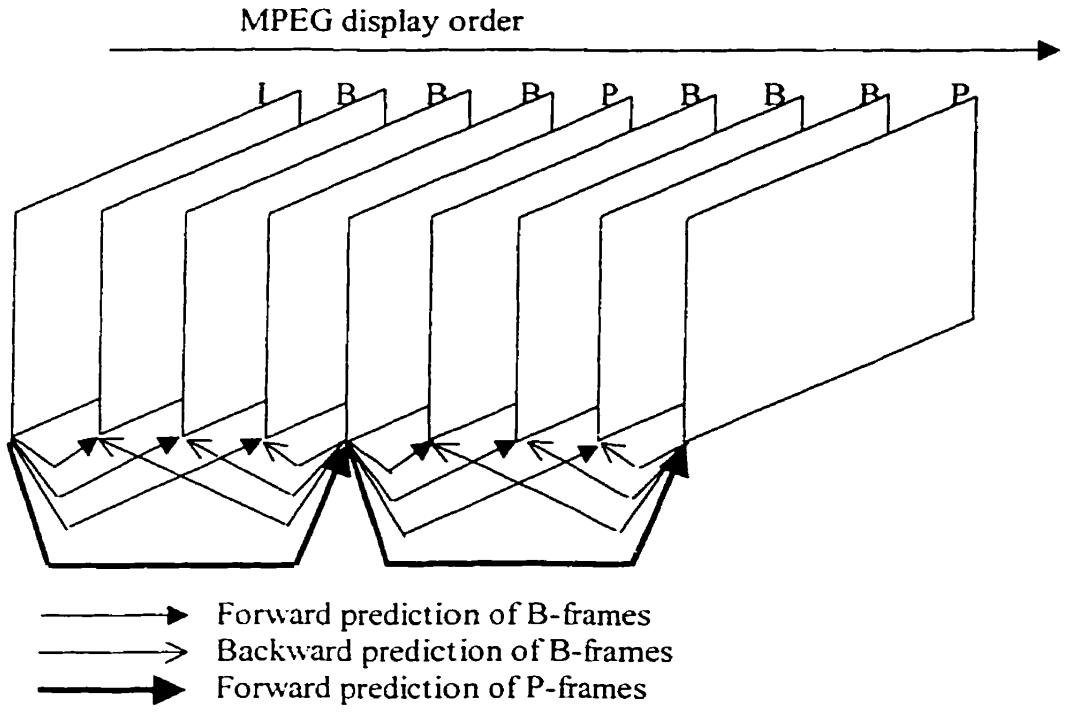


Figure 2-2 Picture types

3) Video Encoder

Figure 2-3 is a diagram showing the essential elements of a video coding system for MPEG –2 Standard. Temporal redundancy is reduced using the following process.

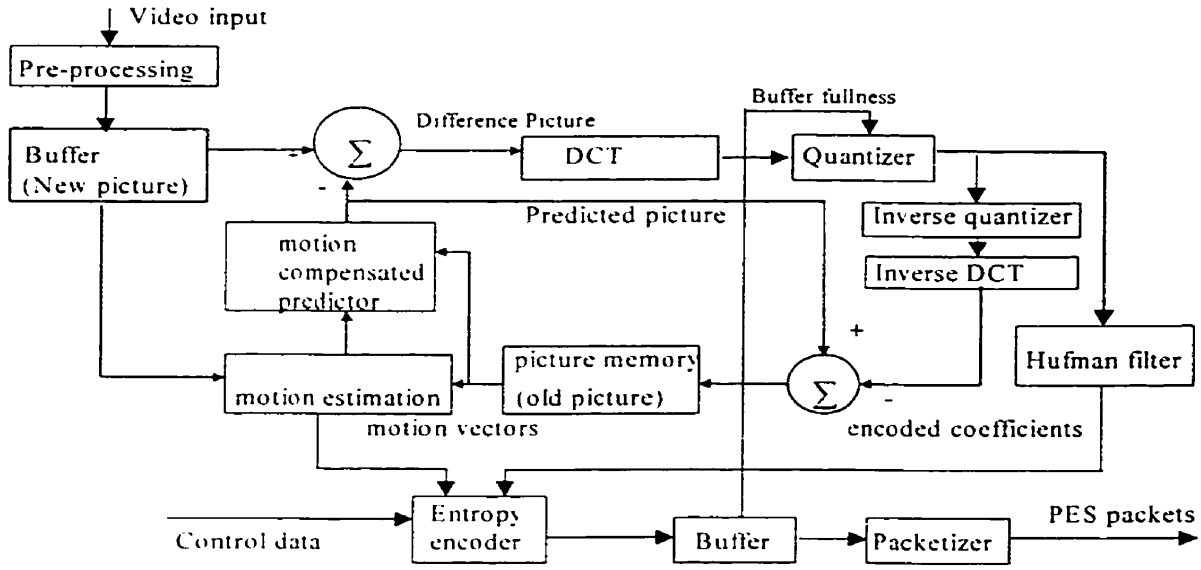
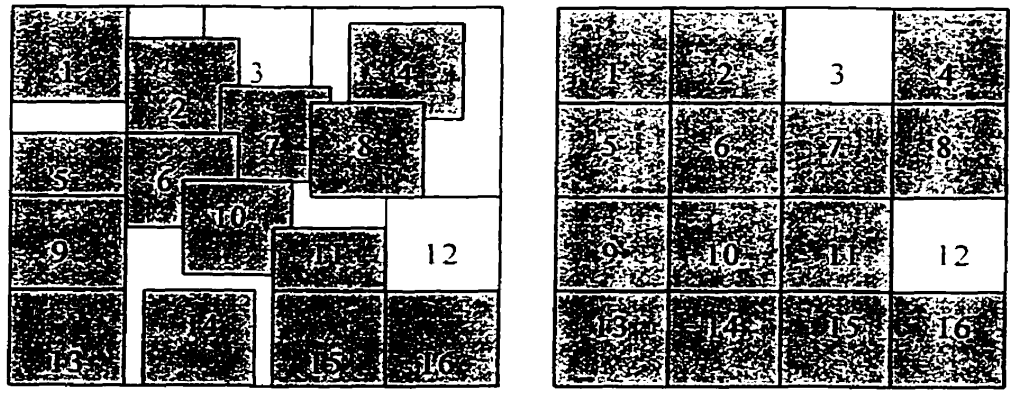


Figure 2-3 Essential elements of video coding system in MPEG –2 standard

In the motion estimation section, an input video frame, called a new picture, is compared with a previously transmitted picture held in the picture memory. Pixel blocks (an area of 16-pixel wide and 16-pixel high) of the previous picture are examined to determine if a close match can be found in the new picture. First, the new picture buffer is divided into 8x8 pixel blocks, each 8x8 pixel block is searched in the old picture area of 16x16 pixels. The match algorithm of motion estimation is: [Eq.2.1]

$$M = \sum_{j=0}^{7}\sum_{i=0}^{7} |A(i,j) - B(i,j)| \qquad (Eq.2.1)$$

Where M is the distortion value, $A[i,j]$ and $B[i,j]$ are the new and old images' pixel values respectively. If the value of M is less than a threshold value, then the vector coordinate of this block is called a close match. When a close match is found, a motion vector is produced describing the direction and distance the pixel block moved. A predicted picture is generated by the combination of all the close matches as shown in Figure2-4. Finally, the new picture is compared with the predicted picture to produce a difference picture [MPG97].



The blocks of a new picture are searched in on old picture

The blocks of old picture predict the new picture

Figure 2-4 Motion compensation

The process of reducing spatial redundancy begin with a DCT (Discrete Cosine Transform) on the difference picture of an 8x8 pixel block. The first value in the DCT matrix (top left corner) represents the DC value of the 64 pixels of the 8x8 block. The other 63 values in the matrix represent the AC values of the DCT with higher horizontal

and vertical frequencies as one moves to the bottom right corner of the matrix. If there is little detail in the picture, these higher frequency values become very small. The DCT values are presented to a quantizer which, in an irreversible manner, can "round-off" the values. Quantization noise arises because coefficients are rounded-offs. It is important that the round off be done in a manner that maintains the highest possible picture quality. When quantizing the coefficients, the perceptual importance of the various coefficients can be exploited by allocating the bits to the perceptually more important areas. The quantizer coarseness is adaptive, and is coarsest (fewest bits) when the quantization errors are expected to be least noticeable. The DCT coefficients are transmitted in a zigzag order as shown in Figure 2-5. After rounding, the higher frequency coefficients often have a zero-value (See Chapter 5 The Algorithm of DCT and IDCT). This leads to frequent occurrence of several zero-value coefficients in sequence.

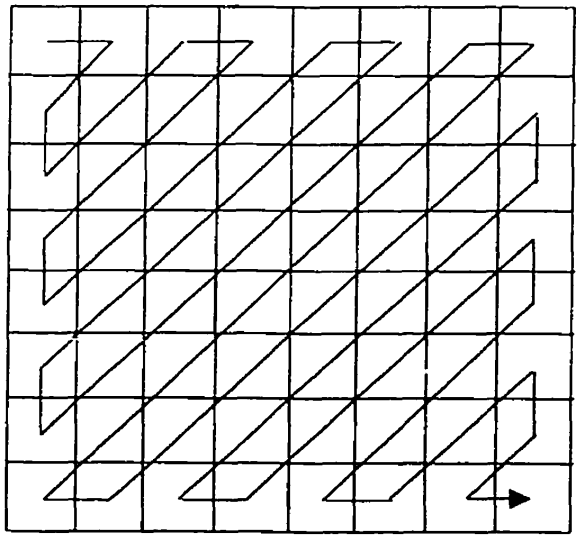


Figure 2-5 Zigzag Scan

The quantizer output is presented to an entropy encoder, which increases the coding efficiency, by assigning shorter codes to more frequently occurring code words. The entropy encoder bit stream is placed in a buffer at a variable input rate, but taken from the buffer at a constant output rate. This is done to match the capacity of the transmission channel and to protect the decoder buffer from overflow or underflow. If the encoder buffer is almost full, the quantizer is signaled to decrease the precision of coefficients to reduce the instantaneous bit rate. If the encoder buffer is almost empty, the quantizer is allowed to increase the precision of coefficients. The output of the buffer is packetized as a stream of PES packets. [DIG94]

In order to use the motion compensated picture for next prediction, the encoder requires the reconstruction of the picture contained in the transmitted bitstream. The quantizer output is presented to the inverse quantizer, then to the inverse DCT. IDCT output adds the predicted picture, and then place the result in the picture memory [MPG97].

The data flow coding system in MPEG standard is shown in Figure 2-6.

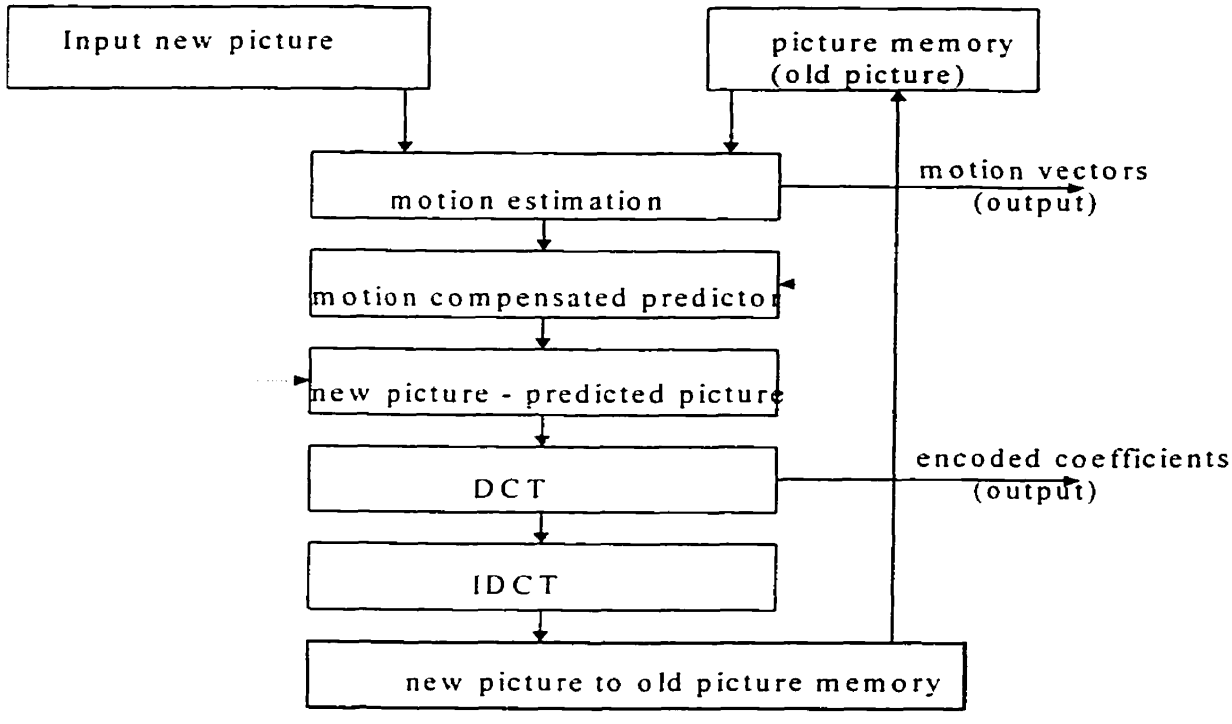


Figure 2-6 Data flow of coding system in MPEG standard

2.1.3 Convolution

Most motion pictures need some pre-processing filter. This pre-processing enhances as perceived by human visual sense. Convolution is one of the popular algorithms used. It will be discussed in chapter 3.

2.2 Motion Estimation Algorithm

Several algorithms for motion estimation have been proposed. A number of popular methods, as well as the one proposed in this thesis, are presented below.

2.2.1 FSM (full search method)

The search of a block frame A (current picture) starts at the upper-left corner of the area of frame B(previous picture). If the value of M in equation [Eq2.1] is less than the threshold value (zero means exact match), stop searching and output motion vector. Otherwise search from left to right and from top to bottom through frame B. The search is stopped when the right bottom corner is reached or when M is smaller than a threshold value. This search sequence is illustrated in figure 2-7. [MPG97]

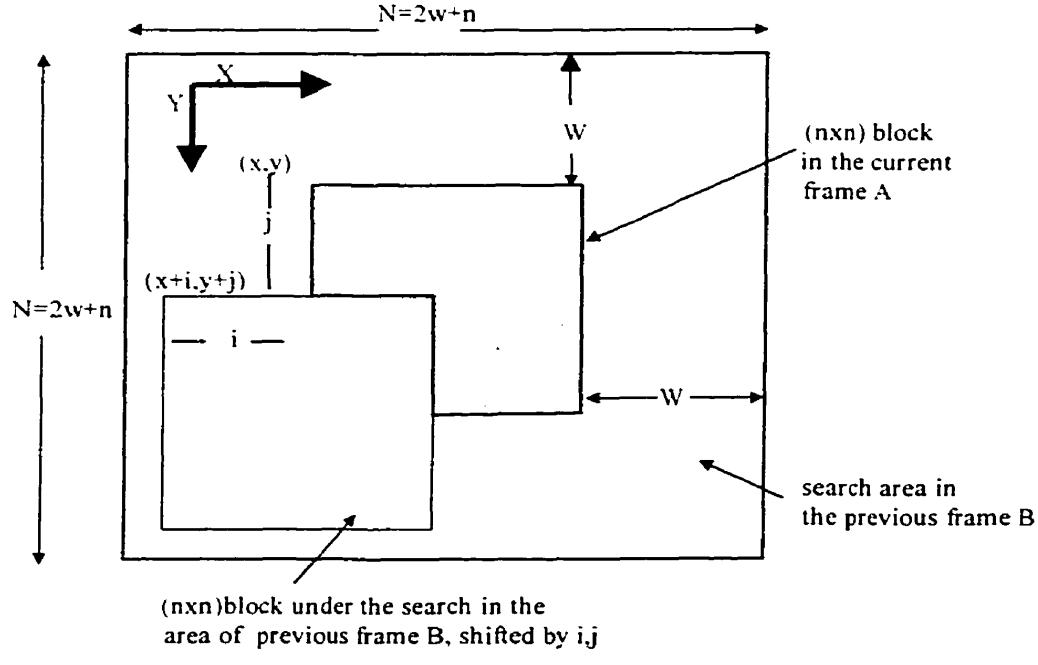


Figure 2-7 The current and previous frames in a search area (N=16,n=8)

2.2.2 CDS (conjugate direction searching)

The search progresses in the direction of the smaller distortion, until a minimum distortion is found (see figure 2-8)[MPG97]. Descriptions of the algorithms refer to points to express the shift between the reference positions in the two compared images. The algorithm is listed below:

M is the value in equation [Eq2.1]; threshold is a value selected by the designer according the allowed error; the left, right, up and down mean the direction of next compared center point from the current center point.

[A]horizontal: compare center point;

if (M<threshold) then stop search and output vector;

else compare left and right point;

if((right(2)<left(0)and(right>threshold))then let right be the new center point;

elsif((right(2)>left(0)and(left>threshold))then let left be the new center point;

endif;

endif;

repeat [A]horizontal until boundary or minimum point is found in horizontal direction;

[B]vertical: compare center point (4)(produced by [A]horizontal);

if (M<threshold) then stop search and output vector;

else compare up and down point;

if((up(7)<down(6)and(up(7)>threshold))then let up(7) be the new center point;

elsif((up(7)>down(6)and(down(6)>threshold))then let down be the new center point;

endif;

endif;

repeat [B]vertical until boundary or minimum point is found in vertical direction;

end;

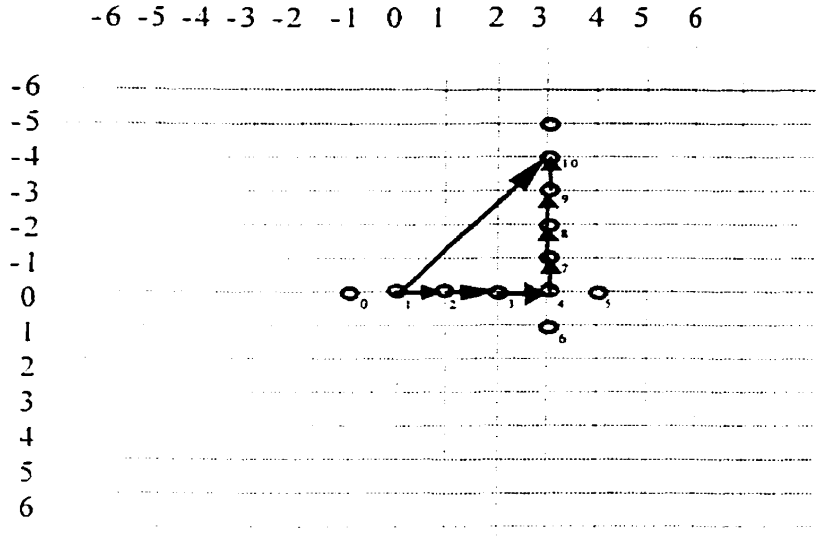


Figure 2-8 CDS method

2.2.3 Three-step searching

The three-step searching method looks for motion displacements. As it progresses through the steps. the search range is decreased. As shown in figure 2-9[MPG97]. The algorithm is listed below:

The definition of M and threshold are the same as with the CDS method.

step(1): compare center point;

if(M<threshold)then stop search and output vector;

else compare four (a) point;

if ((minimum M(a)<threshold)then stop search and output vector;

else compare two M(aa)points(in minimum (a) direction);

if (minimum M(aa)<threshold)then stop search and output vector;

else new center point = the position of minimum M((a) or (aa));

endif;

endif:

endif:

repeat step(1) three times:(minimum M(aa) be the center for b search),(minimum M(b) be the center for c search).

end:

From figure 2-9. the aa point (minimum (M(a) or M(aa))) is as the new center point in step(2) searching. Using a similar method. the b point (minimum (M(b) or M(bb))) is used as the new center point in step(3) searching.

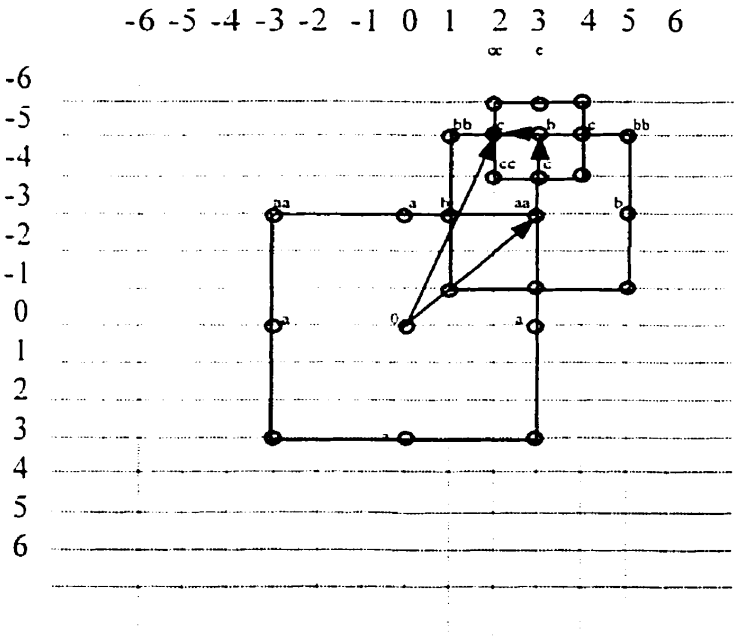


Figure 2-9 Three steps method

2.2.4 CSA (Cross-Search Algorithm)

This algorithm differs from other search methods in the final step. In reference to figure 2-10, the final searching can be either the (X) or (+) directions. This is determined by minimum point. If it is in left up corner or right down corner, the next searching point will choose (X) points. If it is in right up corner or left down corner, the next searching point will choose (+) points [CRO90].

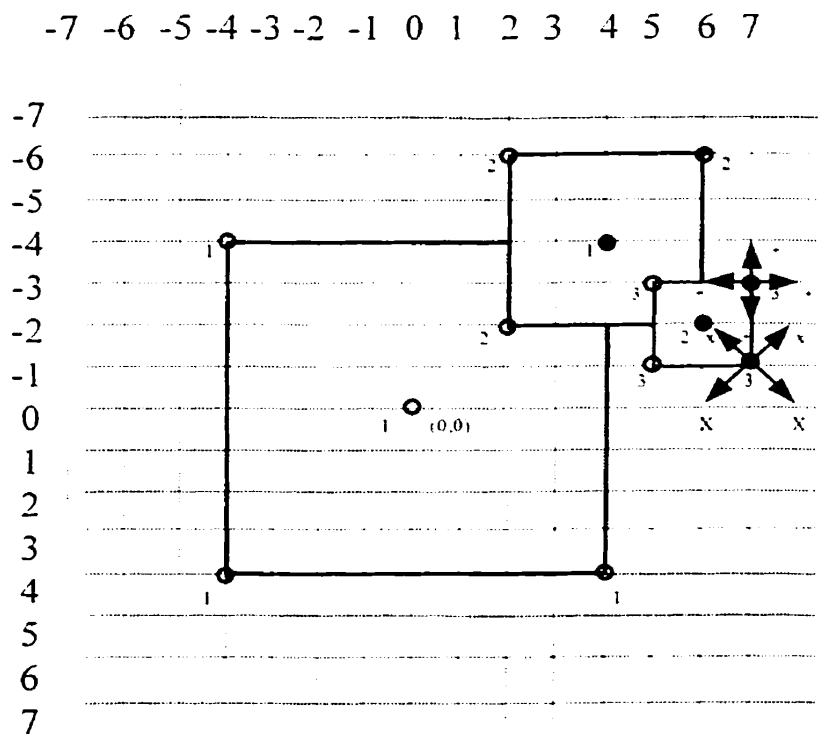


Figure 2-10 CSA method

2.2.5 GFSM(Gradual Full Search Method)

The gradual full search algorithm is a new method proposed here. It begins at the center point and gradually increases the searching range around this point. The method is illustrated in figure 2-11. This method was developed for two reasons.

We analysed the fast algorithms and we found similar problems with most of them. The search direction is usually guided toward the minimum value of M (equation 1) by comparing 4 points at each step. Thus many points inside the search region are skipped. In some cases, moving by one pixel may give very different results. The search direction is controlled by the minimum M, which may lead to incorrect decisions. Fast algorithm are faster than FSM, but they may give incorrect results.

Also, when successive frames do not change much, the last motion sector is a short distance from the center.

The gradual full search method typically takes a short time to find the best match, and yet no point is ignored in the search area. Although the algorithm is slightly complex than FSM, it is much faster for most applications.

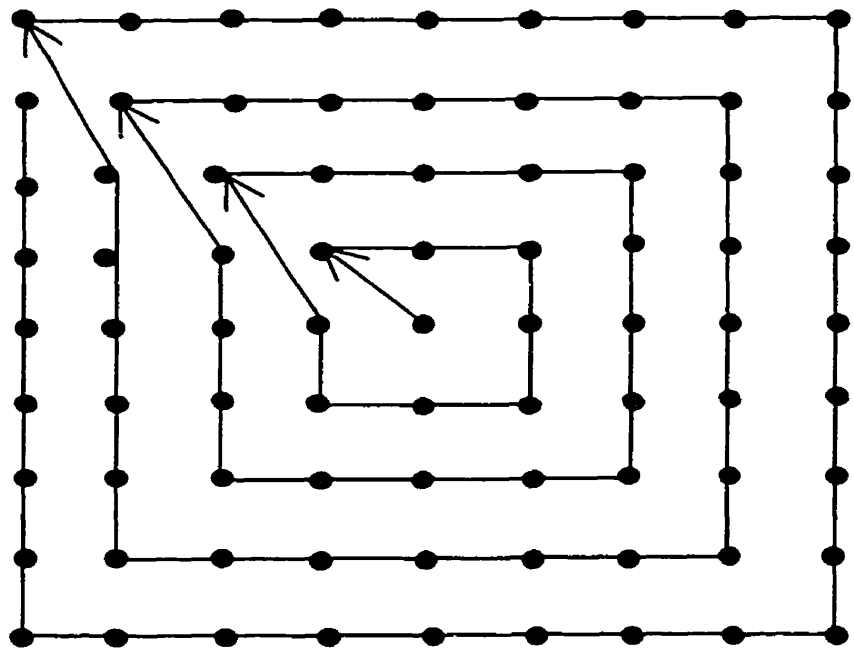


Figure 2-11. Gradual Full Search Method (GFSM)

2.2.6 Comparison of the different algorithms

Table 2-1 shows the maximum number of search points with the different algorithms. The fast algorithms (CDS, CSA and 3STEPS) are faster than FSM and GFSM. When w is large, this is more obvious. However, the fast algorithms sometimes produce incorrect results.

Algorithm	Maximum number of search points	w		
		4	8	16
FSM	$(2w+1)^2$	81	289	1089
CDS	3+2w	11	19	35
CSA	$5+4\log_2 w$	13	17	21
3 step	25	25	---	---
Gradually	$(2w+1)^2$	81	289	1089

W=(size of search area – size of block)/2

Table 2-1 Comparison of different algorithms

2.3 Processor Architecture Review

In order to implement a MPEG coding system, a powerful calculation engine is required. A huge number of calculations are required to perform motion estimation, DCT, etc. Therefore, some special purpose chips are often used to implement these functions. Three different MPEG-2 video encoders are discussed below.

2.3.1 Custom chip set for MPEG-2 coding

The paper "Two-chip MPEG-2 Video Encoding"[TWO96], describes a system composed of two chips that implements a MPEG-2 video encoder. The key features of these chips set are:

- The Enc-M chip mainly executes motion estimation and compensation steps.

- The Enc-C chip is the main coding and control chip. It executes not only coding operations like discrete cosine transformation (DCT), inverse discrete cosine transformation (IDCT), quantization (Q), inverse quantization (IQ), and variablelength coding (VLC), but also header generation, rate control, and output buffer control. It has an external output buffer (FIFO-structured desired from a 2-Mbit DRAM) to meet the requirements of the MPEG-2 algorithm.

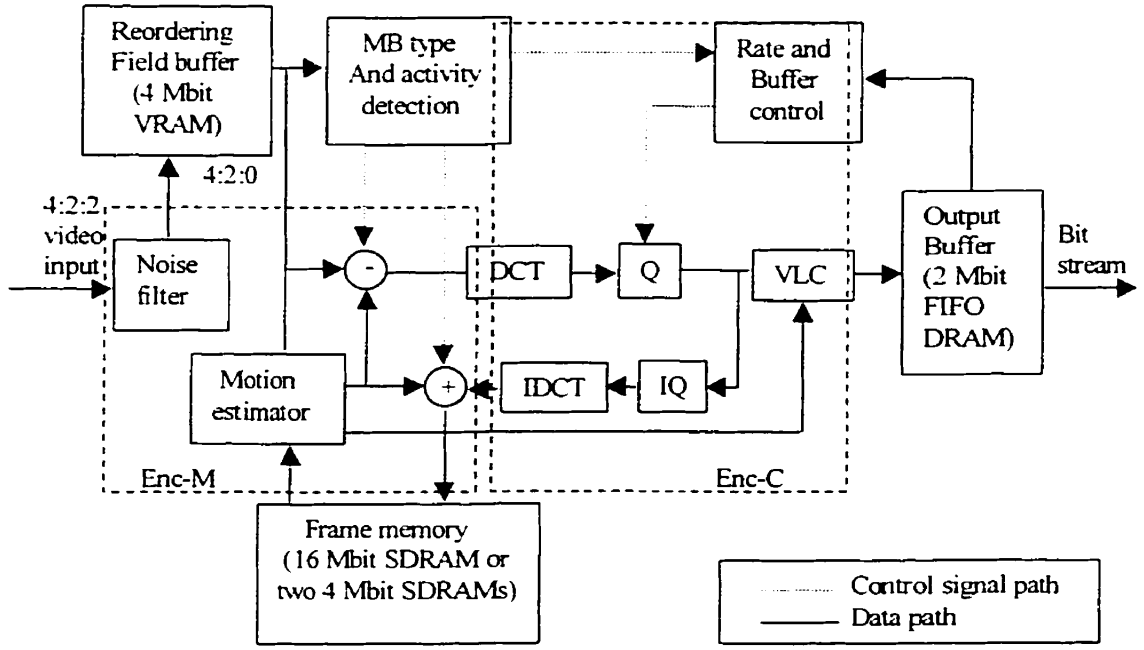


Figure 2-12 Function partitioning in MPEG-2 encoding

Figure 2-12 shows the partitioning of the MPEG-2 encoder. It uses two encoder chips (Enc-M and Enc-c), as well as three peripheral memories, a reordering field buffer, a frame memory, and an output buffer.

Since MPEG-2 is a complex algorithm that requires a flexible and efficient control structure. the pipeline architecture based on RISC CPUs (Figure 2-13) is used. Both the Enc-M and Enc-C have their own RISC CPU. For flexible pipeline operation. each functional unit has a CPU I/O device controlled by the CPU via the I/O port. Some units communicate with neighbouring units in a request-acknowledge manner.

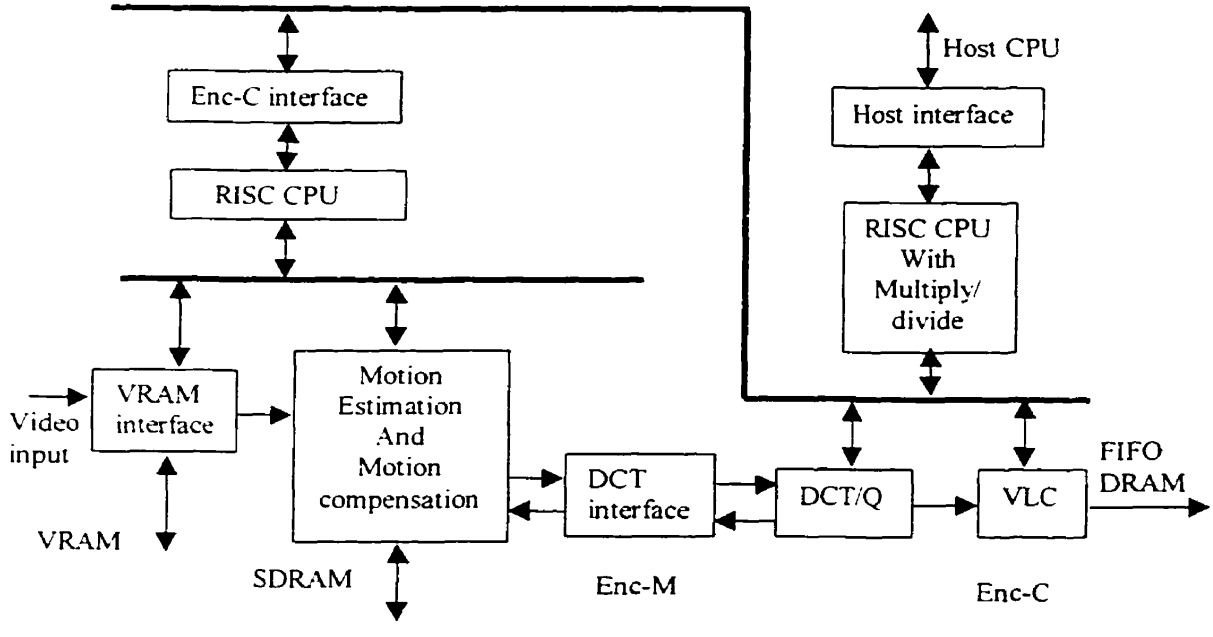


Figure 2-13 Chip set feature of flexible pipeline architecture based on RISC CPUs

The encoder chip set can easily be used to develop a compact encoder system. The encoding algorithm is the MPEG-2 simple profile at main level with a variable frame size from 64 to 720 pixels (column) and 64 to 576 pixels (row). The chip set thus supports the conventional sizes of 720x480. 720x576 and 640x480 pixels required for NTSC, PAL

and VGA standards. Using a 4:2:2 video input format, the maximum frame rate is 30 frames per second for a 720x480 frame size. That means the system processes up to 40,500 macroclocks(16 by 16 pixel) every second for a maximum output of 15 Mbits per second.

2.3.2 VLSI implementation for Motion Estimation

The paper "VLSI Architecture for Motion Estimation using the Block-Matching Algorithm"[VLS96] introduces the EST256 chip used for motion estimation. The architecture, EST256, which consists of 256 processing elements, deals with a search area(32x32 pixel) for block(16x16pixel) and performs 11GOPS at 44MHz clock (subtraction, absolute value determination, accumulation and comparison). Considering a 720x576 pixel image, the processing rate for motion estimation is 49 frames per second.

The number of PEs working concurrently is 256, and each single processor computes the cost function for one of the 256 possible locations of the reference block within the search area. The array outputs the motion vector corresponding to each reference block, 256 cycles after the last pixel of the block has been entered into the array. Figure 2-14 shows the structure of the 256 processors array. To reduce the required bandwidth, EST256 has three 8-bit input ports. After initial latency, the comparator block inputs one error computation in each cycle and compares it with the previous minimum, storing the lowest. The boundary block disables the comparator when its input value is not valid, this condition arises for some locations of the blocks located on the top, bottom, left and right

boundaries of the image. The architecture provides the minimum error value, the coordinates of the motion vector for this position and the error value for the (0,0) motion vector (no movement).

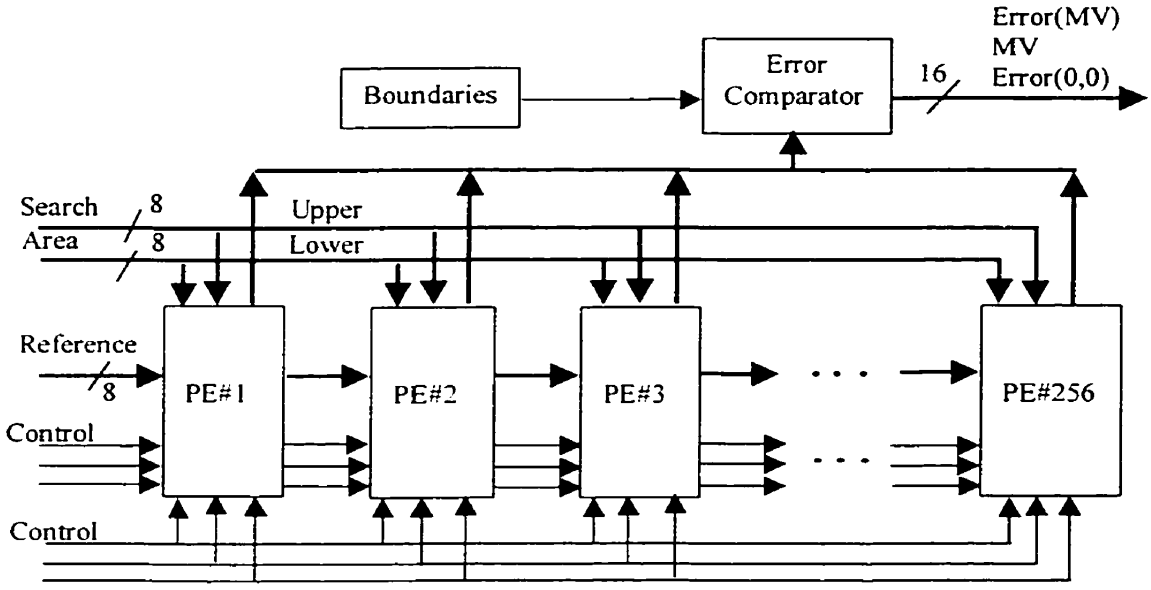


Figure 2-14 EST256 architecture

2.3.3 A PC based image compression system

The paper "A High-performance System for Real-time Video Image Compression Applications"[HIG95] introduces a PC based image compression system. As shown in Figure 2-15, the system consists of a PC-486, a motion estimation processor (MEP), a DCT/IDCT processor (DCT/IDCTP), an image grabber, and a camera. Meanwhile, both the MEP and DCT/IDCTP act as backend processors for PC-486 through its Vesa local-bus interface. The PC-486 handles all the computations except motion estimation, DCT, and Inverse DCT. Currently, by operating at 12.5MHz, the MEP takes around 100us to

compute the motion vector with tracking range 32x32 for each 16x16 block, and, the DCT/IDCT takes around 10us to compute the two dimensional DCT or inverse DCT for each 8x8 block. Also, overlapping data loading and processing can achieve the optimal performance. Therefore, for each 256x256 image frame, the system presented would take around 25.6ms and 10ms for computing motion vectors and two dimensional DCT or inverse DCT individually.

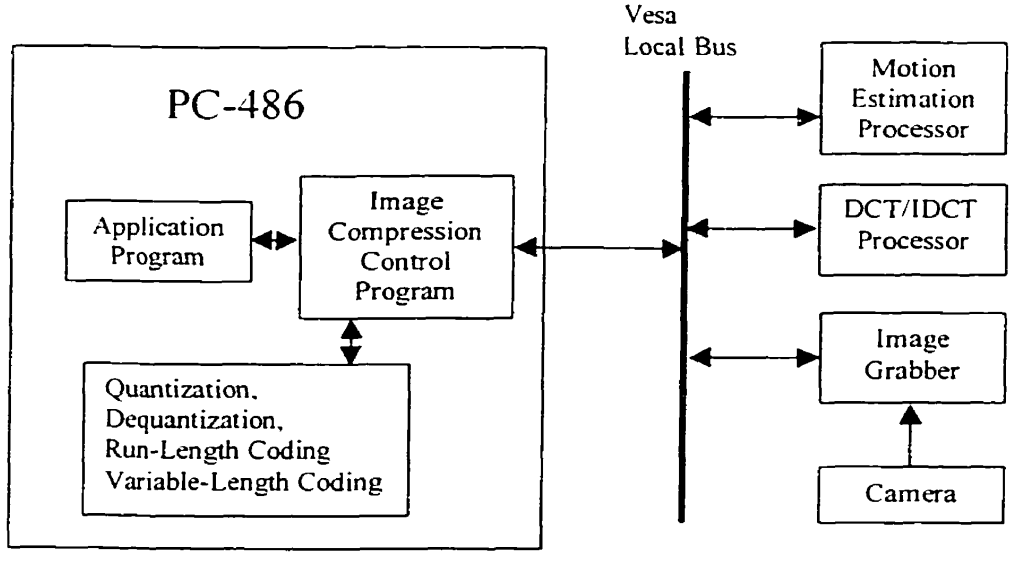


Figure 2-15 Structure Diagram for a Video Image Compression System

Conclusions

As described above, in order to implement real time image processing, special processor or functional units are needed. As a general purpose DSP, single PULSE chip may not be very powerful. However, it is easy to connect multiple PULSE chips to implement real

time image processing. An array of PULSE chips can implement motion estimation, DCT, etc. Each chip also implements a control unit, and these chips can easily be interfaced with other processors. So, PULSE chips give us the ability to build different systems to implement various algorithms and applications.

2.4 SIMD Architecture of PULSE Chip

2.4.1 Introduction

PULSE V1 is a 16 bit, fixed point SIMD array processor designed to operate at 54 MHz. It contains (on a single chip) one controller and four PEs (Processing Elements). The custom designed architecture and instruction set of the PULSE processor allow efficient implementation of all linear (multiply add accumulate etc.), nonlinear (maximum, minimum, medium, rank order, etc.) and hybrid operations, thus providing a complete solution for any fixed point DSP related applications.

PULSE V1 employs heavy parallel operations to handle data I/O, inter-processor communications, address generations, and computations. One parallel instruction could simultaneously provide multiple computations (such as multiply add accumulate, and 3-point rank order), multiple address generations and memory access, multiple data transfer within the PE and between the neighbor PEs. All these operations can be done at the rate of one per cycle however, PULSE data-path is a 4-stage pipeline. One PULSE instruction could perform more than 10 conventional operations. Effectively, the PULSE V1 chip can provide more than 2 Billion RISC like operations per second for linear processing.

Usually conditional execution on SIMD machines can become highly inefficient, since each PE might get different conditions, but the controller can only supply a Single Instruction based on a single condition. The usual way to solve this multi-condition problem is to turn some PEs off, thus wasting computation power. The innovative design of the PULSE V1 processor partly removes these conditional executions by supporting a rich set of nonlinear instructions. For example, each PE can implement a 3 point rank order (maximum, medium, and minimum) in a single cycle of PULSE V1. The implementation of this operation (rank order of 4 vectors, 3 data each) could require more than 60 operations on conventional processors such as TI TMS320C40. In that specified case, the PULSE processor provides more than 4 Billion equivalent RISC like operations per second for nonlinear processing.

To handle real-time image and video processing, PULSE V1 provides up to 864 Mbytes/sec. of bandwidth for data I/O and 432 Mbytes/sec. for inter-processor communications. An innovative communication mechanism provides efficient use of the bandwidth and allows flexible algorithm mapping. Furthermore, the PULSE processor provides a rich set of parallel and vector instructions, which can be used to improve the application performance while reducing the program size.

PULSE V1 provides easy system integration for different classes of applications. It can be used as a stand-alone processor to replace some ASIC chips; it can be used as a co-processor or accelerator to other processors of computer systems; it can have external

programs and data memory for large kernel applications. The cascade of multiple PULSE chips is relatively straightforward, and it can be done without any additional glue-logic. A wide variety of architectures and related system applications can be obtained with suitably cascaded PULSE chips.

2.4.2 Chip Architecture

The PULSE chip version 1 is a PE array, with four data communication ports (two of them are compatible with C40), two address ports, global constant memory and internal program memory. The chip architecture is shown in Figure 2-16. The PE array is the core of the PULSE chip. Its architecture and communication chains are shown in Figure 2-17.

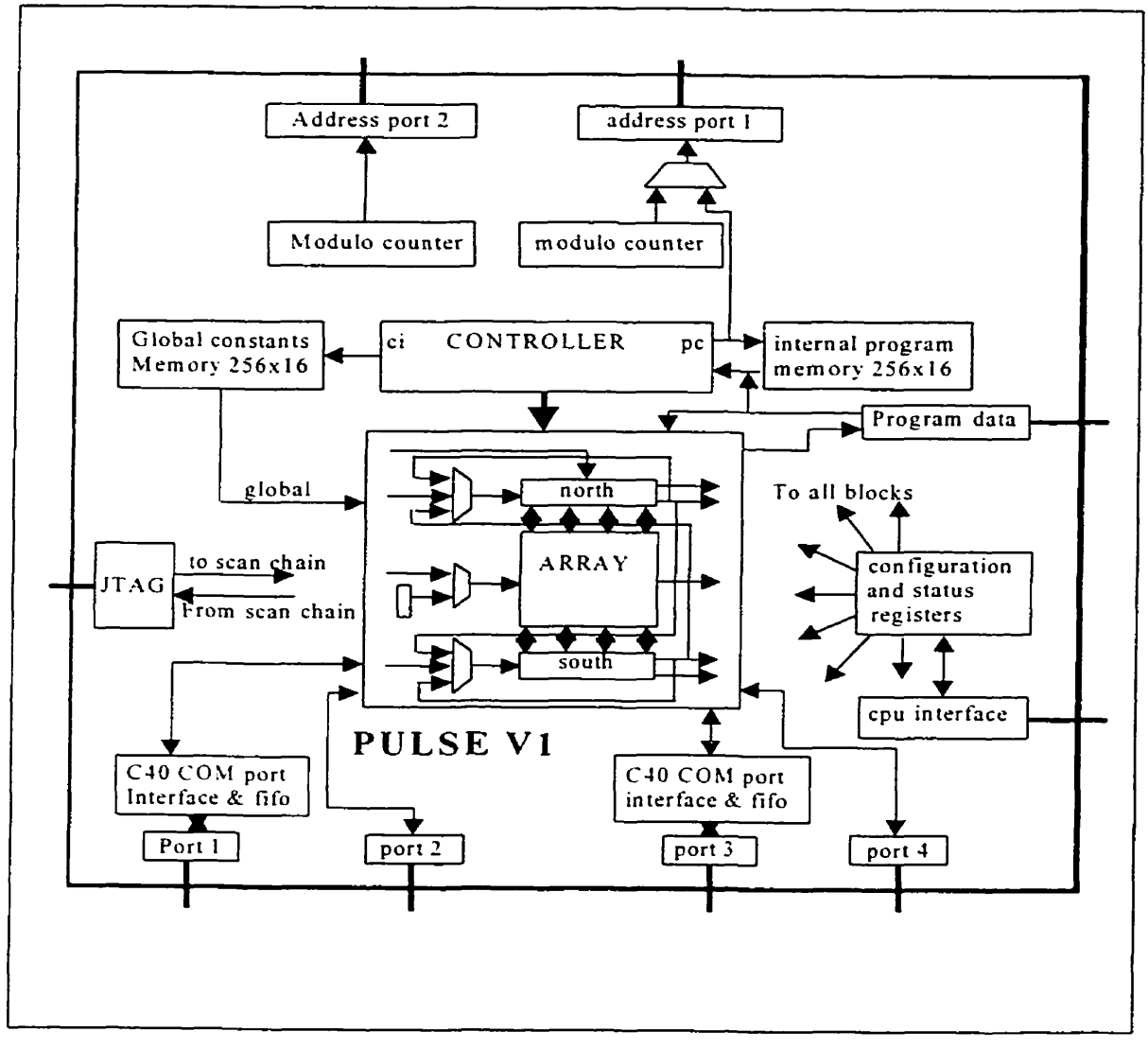


Figure 2-16 PULSE Chip Version 1 Architecture

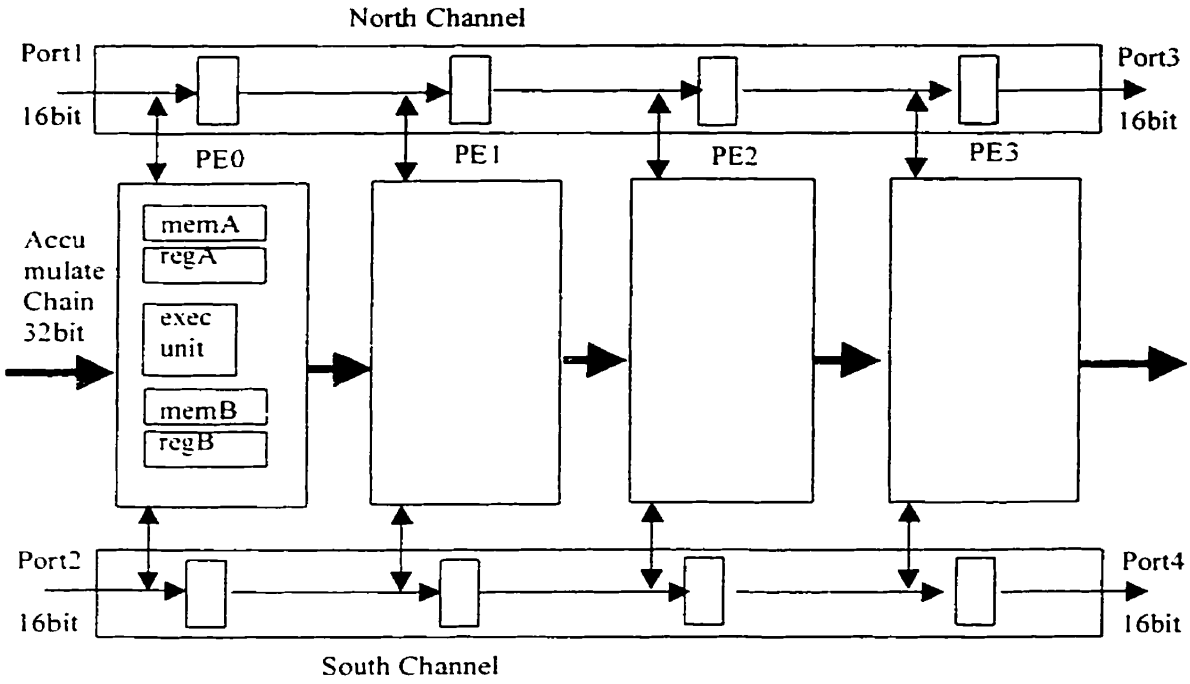


Figure 2-17 Architecture of PEs and Communication Chains

2.4.3 Processing element block diagram

Each of the four PULSE processing elements contains the following elements:

- 2 register files of 32, 16-bit words
  - 1 read port, 1 write port gives optimal storage density/access
- 2 memories of 256, 16-bit words
  - Single port 1 read/write with direct link to communication channels
  - Addressing is direct, register indirect or via two modulo counters per memory for read and write addresses

  - Memories are designed to store data with a longer lifetime than those stored in the register files

- 1 signed multiplier-adder of 16x16+32 bits with 32-bit signed result
  - This is to implement MADD (multiply add)
  - Direct connection of neighbor processors data sources into addend input of the multiplier-adder allows accumulation chain between processors to be built
- 1 accumulator of 32 bits
  - Internal resolution is 33-bit signed with overflow detection
  - Programmable saturation and clipping functions to 32-bit signed or 31-bit unsigned ranges
  - Separate accumulator allows implementation of reduction algorithms it is possible to perform MADDACC – multiply-add-accumulate
- 1 full barrel shifter of 32 bits in, 32 bits out
  - Supports full range of shift logical and arithmetic shift operations
- 1 multi-function 3-operand arithmetic-logic unit
  - Allows single cycle rank, max, med, min, chip and cor functions on three operands
  - Usual arithmetic and logic functions available

The functional units of each processing element are designed to operate in parallel so that a typical PULSE instruction will simultaneously perform a computation, data load and data communication operations. Also, the instruction set is designed to be as orthogonal

as possible so that any operation can be performed on any piece of data, regardless of where it resides. Figure 2-18 presents PULSE V 1.3.f 16-bit processor architecture.

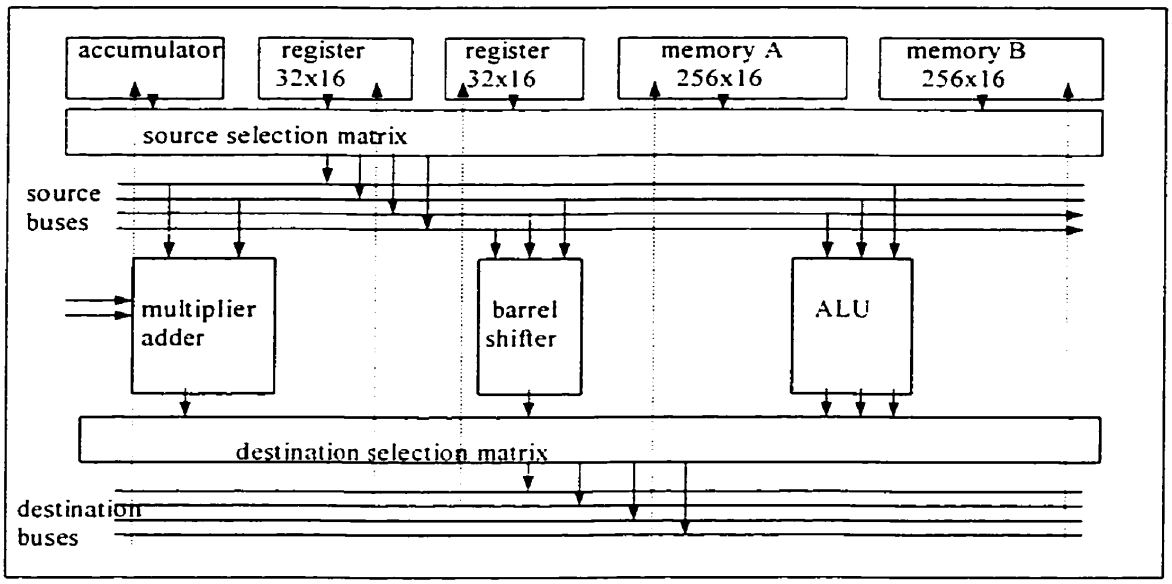


Figure 2-18 PULSE V 1.3.f 16-bit processor architecture

2.4.4 Instruction Set

PULSE employs highly parallel operations to handle data I/O, interprocessor communications, address generations, and computations. Some parallel instructions can simultaneously perform multiple computations, multiple address generations and memory access, multiple data transfer within the PE and between the neighbor PEs. All its operations can be executed at the rate of one per cycle.

PULSE instructions, due to a 4-stage pipeline, generally require four clock cycles, except "stc", "fwd" and "io" instructions (one cycle). The pipeline operation is illustrated in

figure 2-19. At the first cycle, PULSE reads data from memory, register, or port. The second and third cycles execute the operation. The fourth cycle write the results back to memories, registers, or ports.

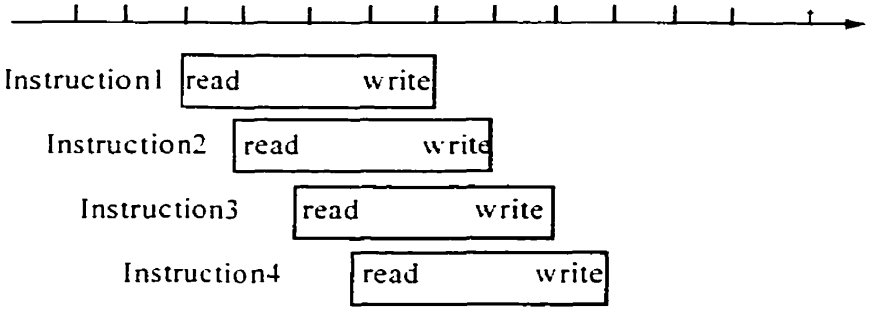


Figure 2-19 Pipeline Structure Instruction in Four Cycles of Clock

The PULSE instruction set is designed for both linear and non-linear digital signal processing, with an emphasis on image/video processing. It provides a rich set of instructions, including conventional instructions and extended non-conventional instructions.

The instruction set is organized into the following functional groups:

- PEs instruction set
- Miscellaneous
- Data movement
- Conventional arithmetic

- Special arithmetic
- Conventional logical
- Shift/rotate
- Transfer of control
- Shift register communication
- Controller instruction set
- Parallel instruction set
- Vector instruction set

Each of these groups is listed in the PULSE Technical Report Composite Document. [PUL96]

2.4.5 PULSE V1 Assembler

The assembler translates assembly-language source files into object files. These files are in common object file format (COFF). Source files can contain the following assembly language elements:

- Assembler directives
- Assembly language instructions

The assembler does the following:

- Process the source statements in a text file to produce an object file.
- Produce a source listing (if requested) and provide the user with control over this listing

- Define and reference global symbols and append a cross-reference listing to the source listing (if requested)

Each time the user uses the assembler, it processes one source program. The source program is composed of one or more files (The standard input is also a file.)

If no file name is provided, the assembler attempts to read one input file from the standard input, which is normally the terminal.

2.4.6 PULSE Applications

As mentioned earlier, the PULSE processor supports both linear and non-linear operations, and allows flexible algorithm mapping. These features make the PULSE processor ideal for a very wide range of applications. Some of them are listed below for reference:

- Filtering
- Transforms
- Image/video/graphics processing
- Image analysis and Machine Vision
- Neural networks
- Speech Processing
- Communications
- Instrumentation

A brief comparison between PULSE V1 and other competitive devices from adaptive solutions, Analog devices, Texas Instruments, and Oxford Computers is provided in appendix A. As can be seen from appendix A, PULSE v1 offers significant advantages in various aspects over the competitive devices, such as strong support for inter-processor communication, and strong support for linear, non-linear and hybrid processing.

The PULSE v1 technical features and the PULSE logic symbol are shown in appendix B and appendix C.

CHAPTER 3

IMPLEMENTING A CONVOLUTION ON PULSE

3.1 The Convolution Algorithm Versus PULSE Architectural Features

The convolution algorithm plays an important role in image processing. For instance, it is used for noise reduction, edge sharpening and skeletonization. The generic convolution is adapted to perform these various functions by appropriately selecting the weights of its kernel. In general, odd size kernels are used.

For example, the 3 by 3 generic convolution algorithm is defined by equation [Eq.3.1].

$$R_{x,y} = \sum_{i=-1}^{1} \sum_{j=-1}^{1} P_{x+i,y+j} \bullet S_{i,j} \qquad (Eq.3.1)$$

In this equation, $R_{x,y}$ is the convoluted pixel, value $P_{x,y}$ is the input image pixel value, and $S_{i,j}$ is the convolution kernel weight. Equation 3.1 indicates that the 3 by 3 convolution P*m,n (m by n pixel image) of each pixel $P_{x,y}$ requires knowledge of the values of its 8 immediate neighbors. On the image boundary, a different algorithm is applied depending on the application.

The following discussion of a convolution applied to a 3 x 3 sample window on a 1K by 1K image will refer to a 2D FIR filter for brevity. These parameters are widely used for preprocessing of images.

1) Each PE in a PULSE chip has two 256 words memory units. For processing 1K by 1K images, they must be partitioned. One possibility is to split the images in vertical bands. Thus we could calculate the 1024 lines of part one, then calculate part two, three and four, as shown in Figure 3-1.

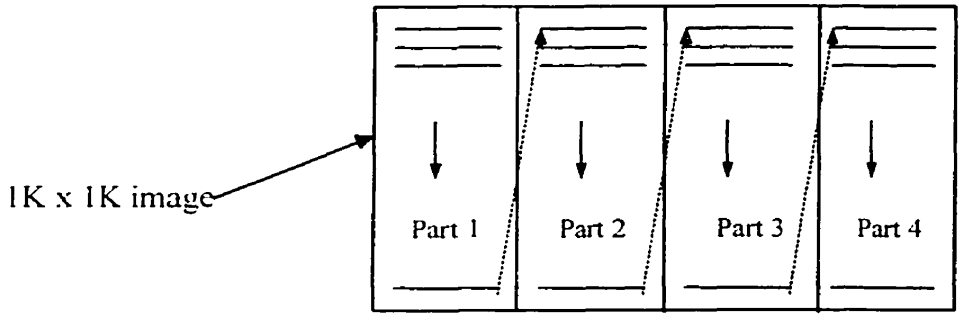


Figure 3-1 Splitting of 1K x 1K original image into four parts for processing on a PULSE chip

2) The processing of the convolution algorithm with a PULSE chip can be executed in parallel, because each chip has 4 PEs. The first data line (256 points) of part one is stored in "memA", the second line is stored in "memB". Data is brought into 4PEs in parallel through the "North Channel"(Figure 2-17). At the first instruction cycle, the data point 1,1 of the original image is stored in PE0, the unspecified data 'x' is stored in PE1, PE2 and PE3 individually. At the second instruction cycle, the point data 1,2 of the original image and the data point 1,1 are respectively stored in PE0 and PE1, while an unspecified

data 'x' is stored in PE2 and PE3. This progresses until the pipeline is full and each processor receives a pixel at each cycle. The resulting distributed data structure is shown in figure 3-2. Respective positions in the rectangle correspond to data stored at the same address of respective processor memories. The processing loop called "loopa" shown in figure 3-5, thus computes 4 pixels of the output image in parallel.

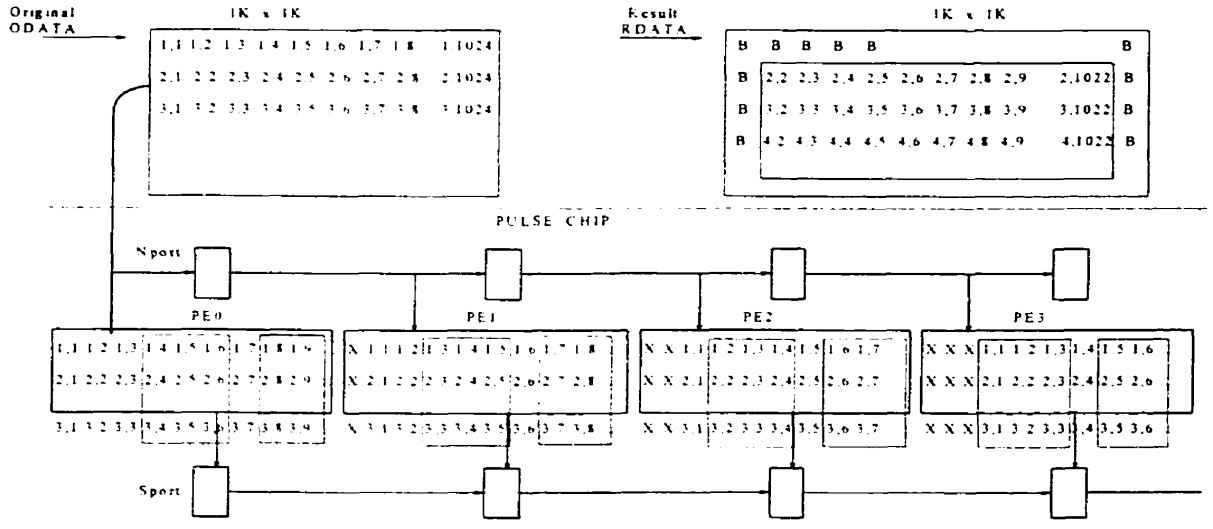
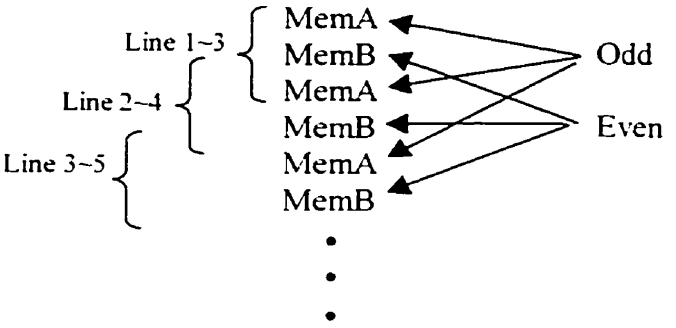


Figure 3-2 Distributed data structure for parallel computation of a convolution

3) Only two lines of data can be stored in memory, because each PE has only two 256 words memory units. The data in "memA" is first used while a third line of data is stored in "memA". When the processing of first data line is finished, the third line has replaced it in memory. The data in "memB" is then used and replaced by a fourth line of data. Thus, "memA" always holds data of odd numbered lines, and "memB" stores even numbered lines. Processing alternates from one to the other.



4) Convolution calculation results stored in external memory. Considering the erosion of image boundaries induced by the algorithm, the output image starts from the second line at the second point (2,2). Therefore, a result picture comprises 1022 by 1022 pixels and edge points "B" as shown in Figure 3-2. The edge points "B" of result image can be filled with the edge points of the original image at the same position. In this case, the edge points are not the result of a convolution. In order to overcome boundary effects between parts (see Figure 3-1), the edge parts are calculated by processing 257 columns. Vertical bands expansion is illustrated in figure 3-3. For part I, columns 1 to 258 (original image) are processed and columns 2 to 257 (result image) are produced. Part II, columns 257 to 514 (original image) are produced and columns 258 to 513 (result image) are produced. Part III, operates on pixels 512 to 770 (original image), and part IV operates on pixels 768 to 1024+2 (original image). Thus, only the edge of the 1K by 1K output picture exhibits boundary effects. There are no boundary effects between part I and part II, part II and part III, and part III and part IV.

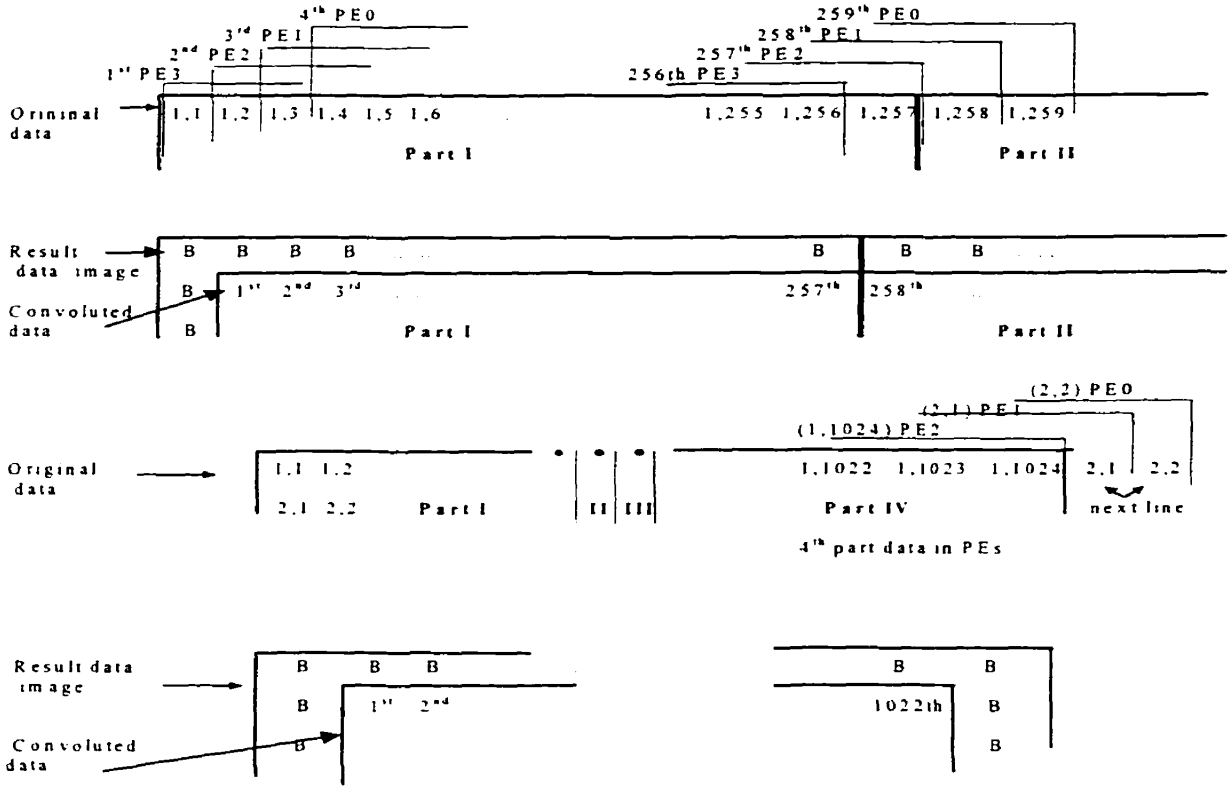


Figure 3-3 Original picture data and result picture data (boundary effect)

3.2 Structure of the Convolution Software

For a 3 x 3 convolution, nine multiplies are needed. The minimum theoretical processing time is 9 multiply instructions and 1 loop control instruction. However each instruction takes 4 cycles. Also, five additional instructions must be inserted with the 9 multiply instructions (three "fwd||nsr||io", one "madd" and one "ld"). Figure 3-4 shows the detail of a loop that computes a convolution.

More precisely, it shows that the result of the first multiply is available after the fifth cycle. Two wait cycles are needed before the second multiply. Then, the "fwd||nsr||io" and "madd" instructions are inserted to avoid waiting time due to processing latency. Using the same principle, two "fwd||nsr||io" and one "ld" instructions are inserted among the remaining instructions. thus, five additional instructions are inserted to fill up various waiting cycle, which reduces the loop execution time.

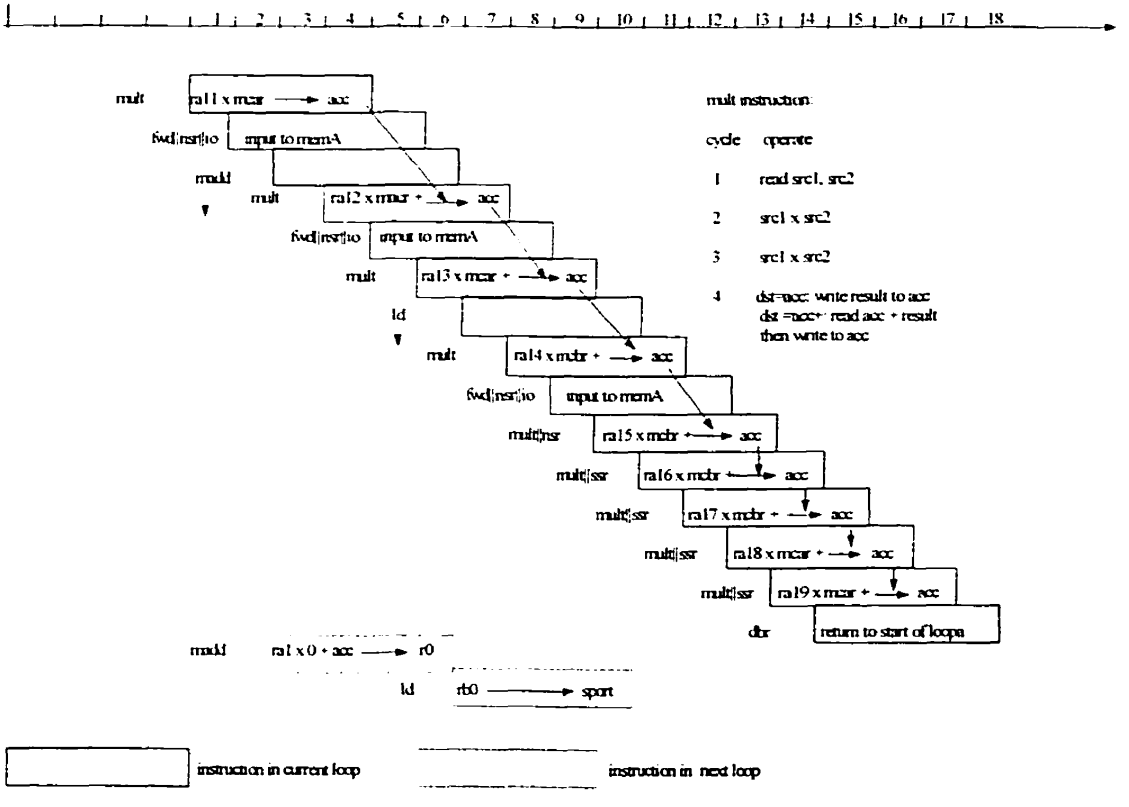


Figure 3-4 Instruction pipelining in a convolution

To process the basic 3 by 3 convolution, the output is first computed and then the result is sent out as shown in figure 3-5. The process of sending out the previously computed output line is overlapped with the calculation of the next to avoid wasting time.

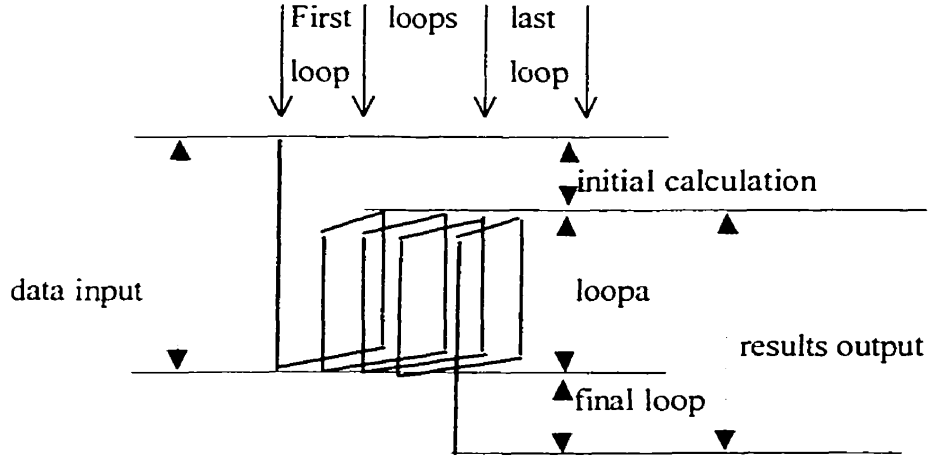


figure 3-5 Overlapping of calculation with exportation of output results

3.3 Summary

An ideal processor with one multiplier requires at least 10 cycles to compute a 3 by 3 convolution. PULSE takes 15 instructions due to pipeline latencies and data dependencies. A PULSE chip computes 4 results at the same time. This corresponds to an average of 3.75 instructions (75ns) per 3x3 convolution. The program flow chart and listing are provided in appendix A. A sample image and the computed results are listed in appendix B.

CHAPTER 4

MOTION ESTIMATION ALGORITHMS AND IMPLEMENTATIONS

The principle of Motion Estimation is to perform a search that maximize the correlation or minimize error between a block in the new (current) picture and a corresponding area in the old (previous) picture. The search process tries to find the coordinate values of a block of already transmitted pixel values in the new picture and transmit them. Thus, if the search succeeds, the block in the new picture is not transmitted.

Motion estimation is a key component of an image processing system such as the MPEG2 standard, because it consumes most of the processing time. For this system, the data rate of 768x480x30x8bps can be reduced 100 times by motion estimation. Obviously, choosing a suitable processor and a good algorithm for motion estimation is key in an image processing system that supports the MPEG2 standard.

4.1 Motion Estimation Algorithm

The motion estimation algorithm that was implemented is given in section 4.1.1. Section 4.1.2 describes the data structure used in PULSE to implement it.

4.1.1 General Description

The mean Square Error (MSE) (Eq.4.1) and Mean Absolute Distortion (MAD) (Eq.4.2) are popular criterions used to measure the fit between data blocks. MAD is the simplest.

$$M = \sum_{i=0}^{\tilde{} } \sum_{i=0}^{\tilde{} } A^2(i,j) - B^2(i,j) \qquad (Eq.4.1)$$

$$M = \sum_{j=0}^{\tilde{} } \sum_{i=0}^{\tilde{} } | A(i,j) - B(i,j) | \qquad (Eq.4.2)$$

The method for choosing the corresponding area in the reference (old) picture is based on the estimation of the moving speed in the content of an image. If this speed is slow, a small area in the reference picture is chosen for searching, otherwise, it is necessary to choose a larger area. The larger the area in the reference picture is, the longer is the search time. Generally, if 8x8 or 16x16 data block in the new picture is chosen, then, 16x16 or 32x32 search area in the reference picture is chosen. Considering the boundary, according to Figure 4-1 and equation (Eq.4.2), there are nine cases of interest, where for each case, the block size is not changed (8x8), but the size of the corresponding area is changed (12 pixels x 12 lines, 16 pixels x 12 lines, 12 pixels x 16 lines and 16 pixels x 16 lines etc.). Thus, nine different programs have been created.

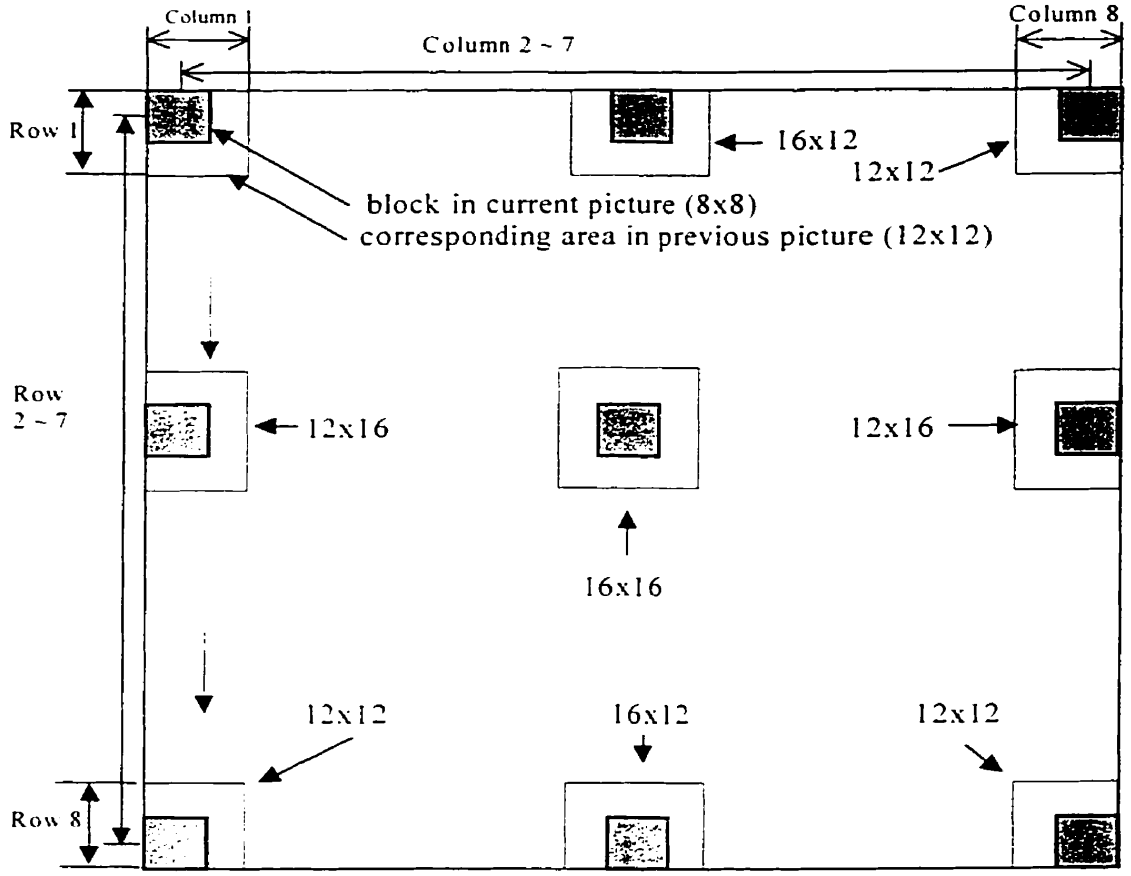


Figure 4-1 Position & relation between blocks & search areas in current picture and previous picture

4.1.2 Data Structure for Motion Estimation in PULSE

The blocks of data from the new picture search area and from the old picture are respectively loaded into memory A and memory B of each PE in PULSE. Each time 128 pixels are loaded. Figure 4-2 illustrates these data structures.

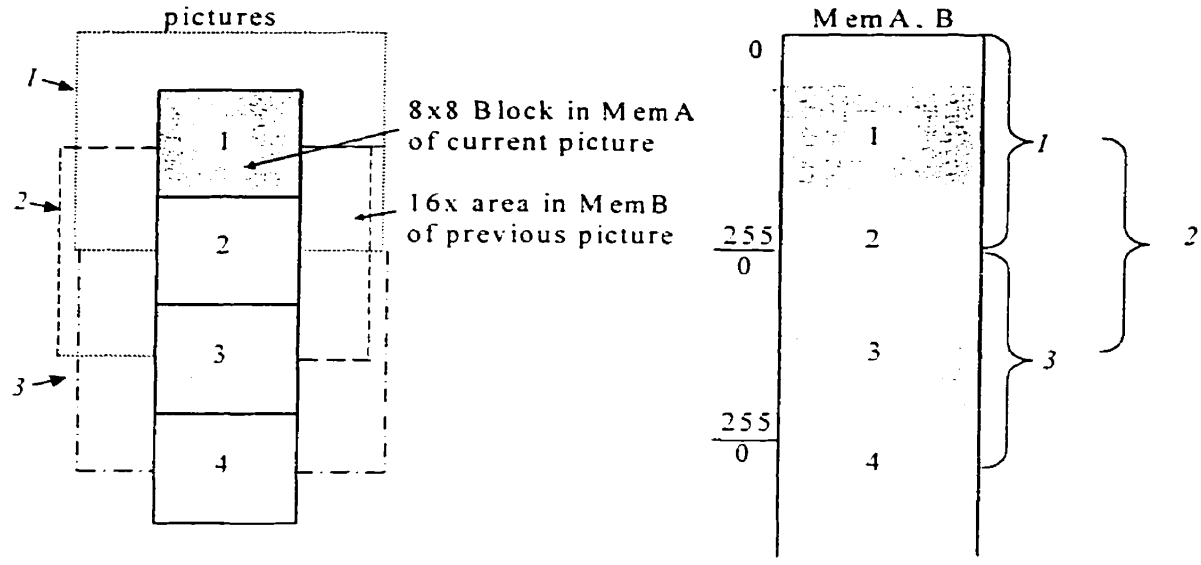


Figure 4-2 Data of blocks and data of search areas in memA and memB

The data is distributed in 4 PEs as shown in Figure 4-3. The results for positions 68, 69, 70 and 71 are calculated in parallel with 4PEs in one processing phase. The next step computes elements 72 to 75. There is no new input data in memA and memB before finishing the block search in the correspond area.

Furthermore, Figure 4-3 illustrates the proposed method to perform a Gradual Full Search Method (GFSM). There are two address counters pointing memory A and B holding respectively a block of new picture and a corresponding area of an old picture. For each comparison, 4 pixel pointers are adjusted to point to the next 4 pixels. When the first search over 64 pixels is completed, the counter indexing memory A is returned to the first pixel unit of the new picture block and the counter indexing memory B is set to the next start pointer of correspond area in the old picture according the gradual search algorithm.

No matter whether a match is found or not. after 81 comparisons with these data sets. the new data of next block and next corresponding area are loaded in MemA and MemB. The next block searching then starts. The gradual search algorithm is somewhat more complex than FSM with respect to the order in which pointers are adjusted. but it is 3 times faster on average.

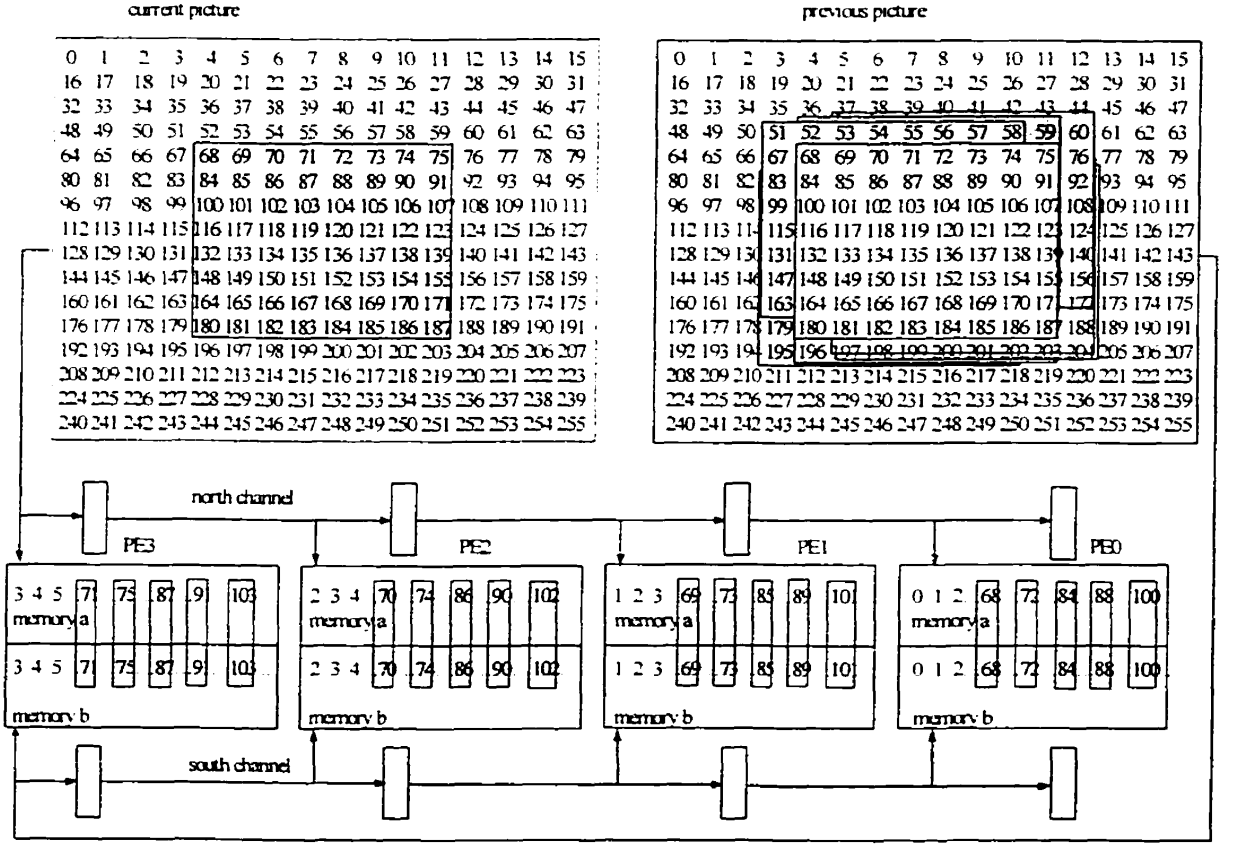


Figure 4-3. Data structure in PEs

4.2 Gradual Full Search Method and Full Search Algorithm with the PULSE Chip

In section 4.2.1, the speed of the GFSM and FSM algorithms (see chapter 2.2) are characterized. Moreover, the basic match program is presented in section 4.2.2.

4.2.1 Speed of GFSM and FSM Algorithms in PULSE

The MAD algorithm needs 192 operations (64 subtractions + 64 absolute value computations + 64 accumulate instructions). If this was spread ideally on 4 processors, a minimum of 48 instructions would be required.

In order to complete 81 block comparisons over one region, 3888 instructions (48 x 81) are needed. There are 5760 (768 x 480 / 64) blocks in one frame. In a real time system, pictures must be processed at the rate of 30 per second. A total of (5760 x 30 = 172800) blocks search must be done every second.

Assuming 54 MHz PULSE chips, (48 x 81 x 172800 / 54E6 = 12.4) 12.4 chips would be required. However, in practice, more resources are needed due to the overhead associated with data input, loop control, address counter setting and instructions that cannot be parallelized.

Figure 4-4 shows the program flow chart and its instructions with GFSM or FSM algorithms, for one block match. For a single search, it uses 768 input + 355 cal. = 1123 cycles or 17.7 cycles / pixel. For a full search, time is 768 input +28009 calculations = 28777 cycles or 449.6 cycles / pixels. Considering that: 1) one PULSE chip can run 54,000,000 instructions/second, 2) one 768x480 pixels image has 96x60=5760(8x8)

blocks to be processed out. 3) 30 frame pictures/ second are used. then 28777instructions x 30 x 5760 / 54E6 = 92 PULSE chips are needed for GFSM.

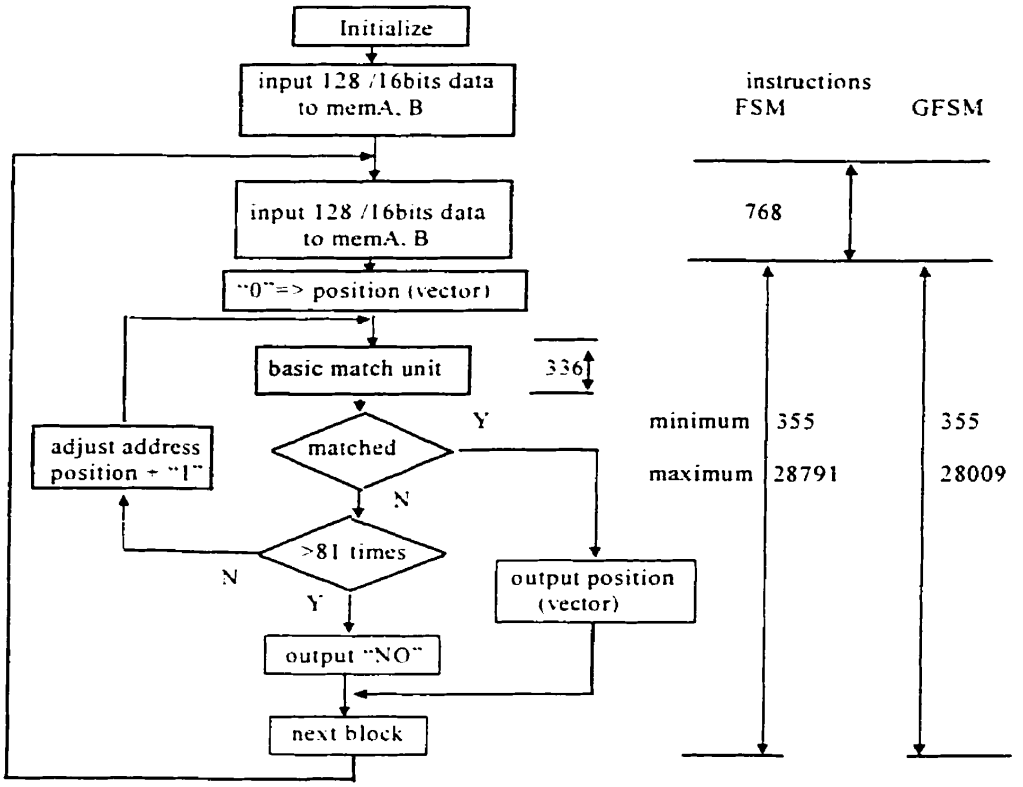


Figure 4-4. FSM & GFSM program diagram and its instructions (one block)

4.2.2 Motion Estimation Program for PULSE

Figure 4-5 shows the instructions executed to perform one basic match program for the full search method (FSM) or the gradual full search method (GFSM). (corresponds to PULSE assembly language)

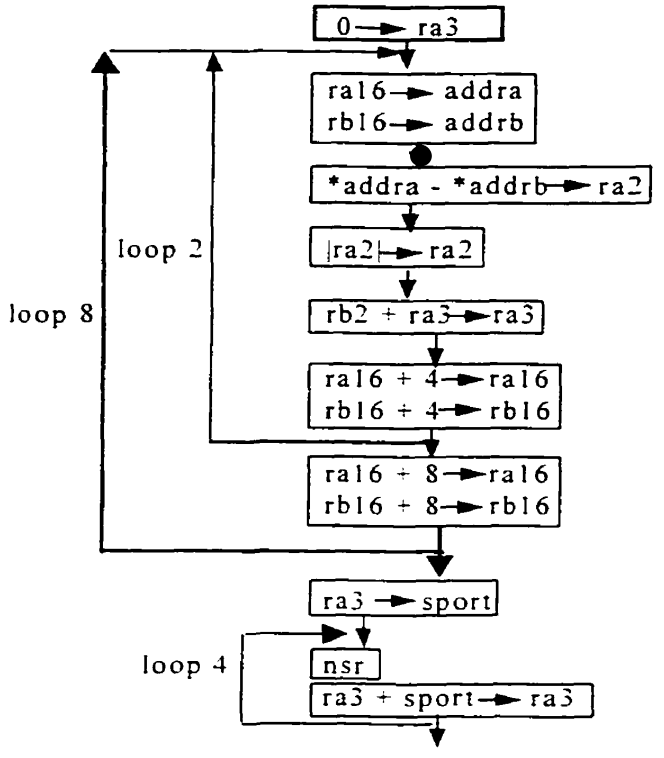


Figure 4-5. Flowchart of a basic match unit

As shown in figure 4-2, address pointer for memA and memB is 256. If the address pointer is over 255, it will return to 0. For the second block search, some data remains from the first block search, thus the start address in the new picture is 194 and the start address of area in the old picture is 128. In other words, each block search only needs to renew 128 pixels. In order to implement this behavior, mechanism to adjust address pointers are needed. In the basic match program, 72 instructions are required for this task. For one block search, 5832 instructions are needed, out of which a fifth are used to check and adjust address pointers. These instructions are identified in basic match program (Figures 4-7 and 4-8).

For each block search, Figure 4-6, if 256 memory locations are reserved, starting at address zero, then some instructions can be saved. In order to implement this idea, for each block search, 256 pixels (one search area) must be transferred from external memory. As shown in Figure 4-6, some instructions are added to adjust the start address of the pointer that accesses the external memory. 80 PULSE chips would be required to support real time processing of this algorithm.

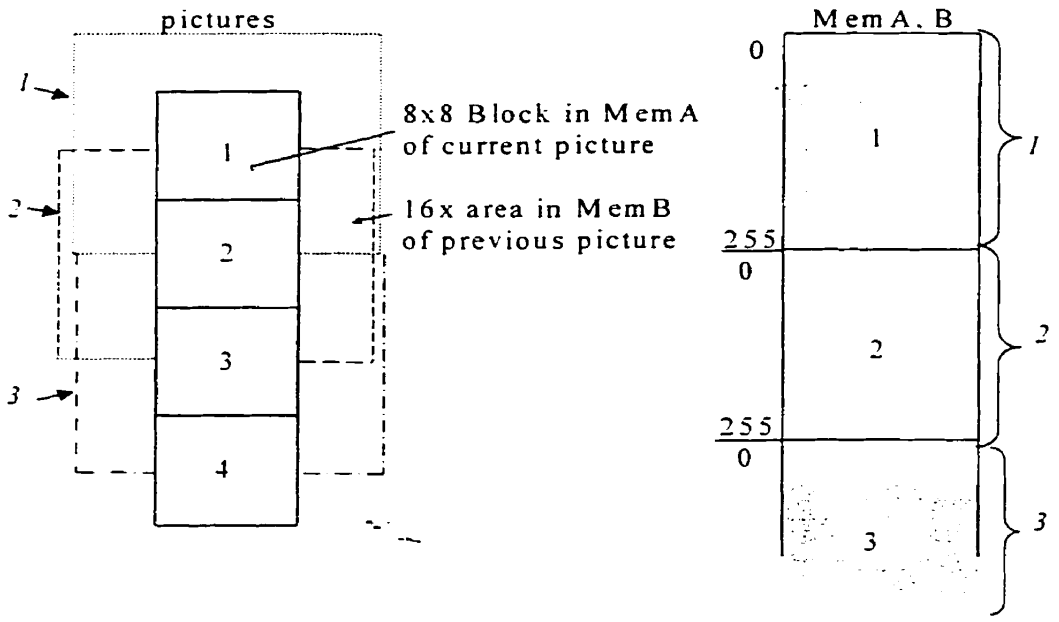


Figure 4-6 Data of block search in independent memory space

In figure 4-7, the basic match program called 't1', contains some 'nop' instructions due to PULSE' s pipeline architecture. In order to reduce the number of 'nop' instruction, some instructions can be shifted without changing the result. In Figure 4-8, the program 't2' is an improved version exploiting this idea. It removes loop2 to save loop set instruction cycles. The marked instructions are moved into other places occupied by 'nop' in 't1'.

With this improved version 21300instructions x 172800/54E6 = **68** PULSE chips are needed.

```
Program 't1'-------------------------------------------------
t1:         ld 0, ra3;  clear accumulater ra3
            ld rb6, rb1
            push 8;             accumulate 64 pixels
      loop3:
            push 2;      accululate 1 line (8 pixels)
      loop2:
            #2 nop
            ld ra16, addra
            ld rb16, addrb
            #3 nop
            sub *addra, *addrb, rb2;ra2
            #3 nop
            abs rb2, rb2;
            #3 nop
            add rb2, ra3, ra3
            add ra16, 4, ra16;       +4
            [redacted]
            add rb16, 4, rb16;       +4
            [redacted]
     dbr loop2
            add ra16, 8, ra16;       16-4x2=8
            [redacted]
```

```
        add rb16, 8, rb16;      16-4x2=8
        [illegible]
    dbr loop3
        ld ra3, sport
        push 3
    loop1:
        add ra3, sport, ra3
        ssr; add before ssr
    dbr loop1
        sub ra16, 128, ra16;    -128 back to start point
            iflt ra16, 0, 0; (if ra16<0, +256)
            #3 nop
            add ra16, 256, ra16
            else
            nop
            restore
        sub rb16, 128, rb16;    -128 back to start point
            iflt rb16, 0, 0; (if rb16<0, +256)
            #3 nop
            add rb16, 256, rb16
            else
            nop
            restore
ret
** The "and ra16, coffh, ra16" instruction keeps the value of ra16 less
than 256, when increase 4.
        add ra16, 4, ra16;      +4
```

```
        and ra16, 00ffh,ra16;   model 256
```

Figure 4-7 Program t1

program 't2'----------

```
tt1:    ld 0, ra3;  clear accumulater ra3
        ld rb6, rb1
        push 8              ;accumulate 64 pixels
loop3:                      ;accululate 1 line (8 pixels)
        ld ra16, addra
        ld rb16, addrb
            nop
            add ra16, 4, ra16;  +4
            add rb16, 4, rb16;  +4
        sub *addra, *addrb, rb2;ra2
        nop
        ld ra16, addra
        ld rb16, addrb
        abs rb2, rb2;
        #3 nop
        add rb2, ra3, ra3
        sub *addra, *addrb, rb2;ra2
        #3 nop
        abs rb2, rb2;
            [redacted]
            [redacted]
            [redacted]
```

```
            add rb2, ra3, ra3
      dbr loop3
            ld ra3, sport
            push 3
      loop1:
            add ra3, sport, ra3
            ssr; add before ssr
      dbr loop1
            sub ra16, 128, ra16;     -128 back to start point
            sub rb16, 128, rb16;     -128 back to start point
ret
```

Figure 4-8 Program t2

Sample programs and corresponding results are provided in appendix A. The simulation uses Mentor Graphic QHSIM tool. The source file includes two 64x64 pixel images. The 8x8 block FSM and Gradual Full Search Method (GFSM) are implemented with 16x16 matching area. The running time. matched pixels coordinate values in new picture and the coordinate values of corresponding area in old picture for first block searching are also provided in appendix A.

CHAPTER 5

DCT & IDCT ALGORITHMS AND IMPLEMENTATIONS

5.1 DCT & IDCT Algorithms

In this section, a data compression algorithm, the Discrete Cosine transform (DCT) [FAS87] and its inverse algorithm, the Inverse Discrete Cosine Transform (IDCT) [IEE90] are described. These algorithms are widely used in image compression programs. They implement a transform from the time domain to the frequency domain.

5.1.1 DCT

DCT is an essential part of the MPEG data compression. There are two good reasons for using DCT in data compression. First, DCT coefficients have been shown to be relatively uncorrelated, and this makes it possible to construct relatively simple algorithms for compressing the coefficient values. Second, the DCT is a process for (approximately) decomposing the data into underlying spatial frequencies. This is very important in terms of compression, as it allows the precision of the DCT coefficients to be reduced in a manner consistent with the properties of the human visual system. [MPG97]

For data compression with the MPEG standard, two-dimensional array (2-D) of samples are considered. Arrays of eight points by eight points (8x8) are usually considered. Suppose that a 2-D array of sample, f(x,y), is to be transformed into a 2-D DCT. The equation is as below (Eq.5.1):

$$F(u,v) = C(u)C(v)/4\sum_{y=0}^{7}\sum_{x=0}^{7} f(x,y) \bullet \cos[(2x+1)u\pi/16] \bullet \cos[(2y+1)v\pi/16]$$

$$C(u) = 1/\sqrt{2} \quad if \quad u = 0; \quad C(v) = 1/\sqrt{2} \quad if \quad v = 0 \qquad (Eq.5.1)$$

$$C(u) = 1 \quad if \quad u > 0; \quad C(v) = 1 \quad if \quad v > 0$$

For example, the input value of matrix f(x,y) is:

120 108 90 75 69 73 82 89

127 115 97 81 75 79 88 95

134 122 105 89 83 87 96 103

137 125 107 92 86 90 99 106

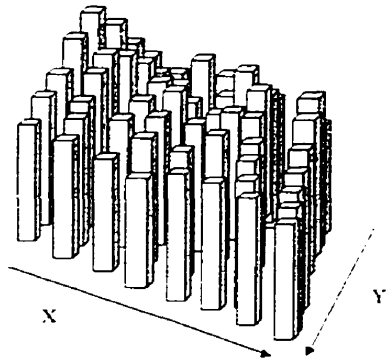
131 119 101 86 80 83 93 100

117 105 87 72 65 69 78 85

100 88 70 55 49 53 62 69

89 77 59 54 48 42 51 58

Also, suppose '0' means black and '255' means white. The corresponding 8x8 bar diagram is represented below.



Then, the DCT transform calculated with equation *(Eq.5.1)*, produces the following result:

700 90 100 0 0 0 0 0

90 0 0 0 0 0 0 0

-89 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

It is remarkable that almost all values are equal to zero, the non-zero values are concentrated at the upper left corner of the matrix. These non-zero values are transferred to the receiver in zigzag scan order (see Chapter 2, Figure 2-5), which is 700 90 90 –89 0 100 0 0 0 ... 0. The zero values are not transferred. They are replaced by an 'end-of – block' symbol.

5.1.2 IDCT

IDCT is a inverse algorithm of DCT. It is used to regenerate the data back to the time domain from frequency domain representation. The IDCT algorithm is expressed in equation *(Eq.5.2)*.

$$f(x,y) = \sum_{v=0}^{7}\sum_{u=0}^{7} C(u)C(v)/4F(u,v) \bullet \cos[(2x+1)u\pi/16] \bullet \cos[(2y+1)v\pi/16]$$

$$C(u) = 1/\sqrt{2} \quad if \quad u = 0; \quad C(v) = 1/\sqrt{2} \quad if \quad v = 0 \qquad (Eq.5.2)$$

$$C(u) = 1 \quad if \quad u > 0; \quad C(v) = 1 \quad if \quad v > 0$$

In equation *(Eq.5.2)*, the F(u,v) of DCT result with *(Eq.5.1)* will be transformed again, the result of matrix f(x,y) is as below:

120 108 91 75 69 73 82 89

127 115 97 81 75 79 88 95

134 122 105 90 83 87 96 103

137 125 107 92 86 90 99 106

131 119 101 88 80 83 93 100

117 105 87 72 65 69 78 85

100 88 70 59 49 53 62 69

89 77 59 54 48 42 51 58

Comparing the original matrix f (x, y) and the inverse transfer matrix f'(x, y), there are few small differences between them. It caused by the calculation accuracy. In other word, it is limited by the word length of the processor. This error is generated by two times transfer calculation (DCT and IDCT). It means that the error is produced at remote decoder device. In order to limit error accumulation in the decoder, the IDCT can be used in encoding system to generate an 'old picture' (See Chapter 2, Figure 2-3).

5.2 Implementation of DCT & IDCT on PULSE

This section deals with the storage of a cosine table and to the time required to compute it.

5.2.1 Data Structure of DCT on PULSE

Parallel processing can be used to compute DCT and IDCT. With PULSE chips, a cosine table is inserted in memA. The use of this cosine table can save a lot of calculation time. For example, the value of (cos[(2x+1)uπ/16]) can be found in the cosine table according the position of (x, u). For a 8x8 DCT or IDCT, a 64 elements cosine table can be generated with a C++ program executed on a host and then loaded to PEs memories. Here, a benefit of using a C++ program on a host rather than the 16bit PULSE chip is that values are more exact. The cosine table size is related to the window size (8x8 pixel or 16x16 pixel). So, this method is very effective for processing fixed window sizes.

Figure 5-1 shows the cosine table and elements stored in PEs memories continuously. The data structure is the same in each PE. In order to calculate in parallel, the beginning positions for 'u' in 4 PEs are 0, 8, 16 and 24 individually (See Figure 5-1 in shadow).

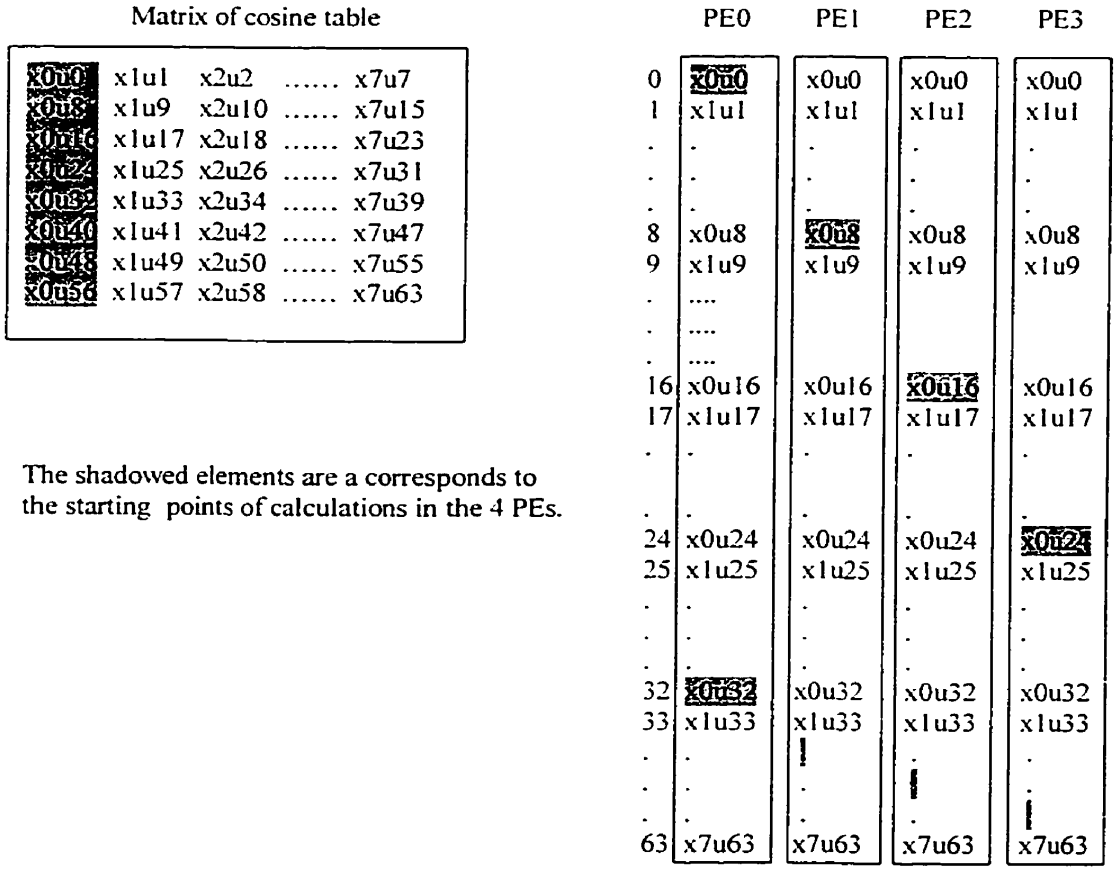


Figure 5-1 cosine table and input data stored in PEs memories

Figure 5-2 shows the program flowchart and the associated instruction count. From inside to outside, there are four loops (loopx, loopy, loopu and loopv). The calculation 'f(x, y)*cos[(2x+1)*u*3.14/16]*cos[(2y+1)*v*3.14/16]' requires only two instructions 'mult' and 'macc' in loopx. Using cosine table and setting address pointer appropriately makes

the complex calculation very simple and parallel. The key idea is that the address pointer points to different units in each PE. It is shown in Figure 5-2 (dark shaded block).

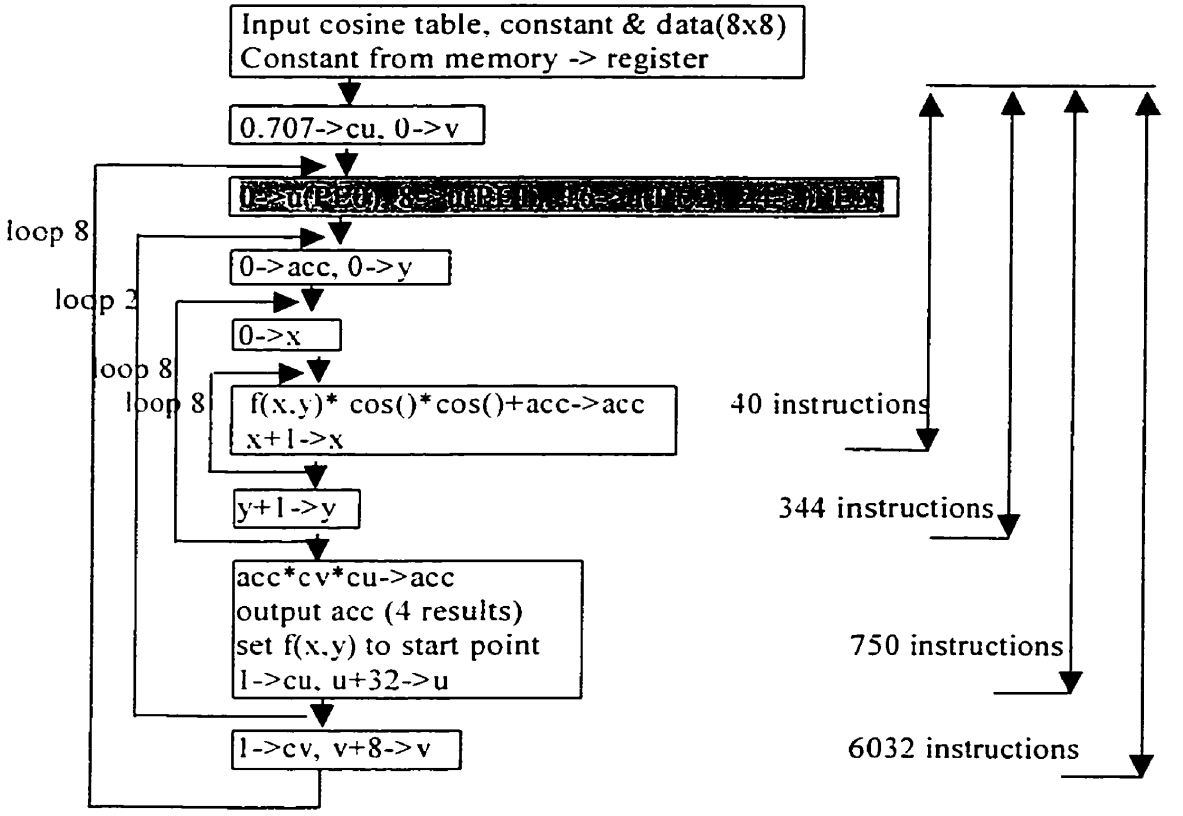


Figure 5-2. DCT Program Diagram and Instructions

Appendix H and I, provide an implementation of DCT and IDCT assembly programs for the PULSE machine, as well as the content of the cosine table, and compared C++ programs of DCT and IDCT.

5.2.2 Requirements and performance for DCT and IDCT on PULSE

The flowchart of a DCT program is presented in Figure 5-2. For each DCT in an 8x8 block, 6032 instructions are needed. To support real time DCT 6032 instructions x 172800 / 54E6 = **19** PULSE chips are needed. Since the IDCT has the same complexity, the DCT and IDCT for the MPEG codec requires 38 chips, which is half the number of chips required for motion estimation (68 chips). Obviously, motion estimation, DCT and IDCT represent the whole complexity of a MPEG codec.

CHAPTER 6

IMAGE PROCESSING WITH PULSE CHIPS AND A C40 PROCESSOR

A PULSE chip is a parallel SIMD processor. It is not very high efficient when executing code with conditionals. For example, when an "if" instruction is encountered, the operation is modulated by four conditionals on 4 PEs. It is generally very difficult to predict which condition from each PE influences the result. In order to get a certain condition from a PE, suppose the jump condition from PE0, which is set active "0", the PE3, PE2 and PE1 are set inactive "1" with instruction (ldcr 1110b, acm), then PE3, PE2 and PE1 are set active(ldcr 0000b, acm) after a jump instruction. The following program illustrates this method:

```
        ldcr 1110b, acm
        ifeq ra3, 0, 0
        #3 nop
        bpa jump1
        ldcr 0000b, acm
        :
        :
jump1: nop
        ldcr 1111b, acm
```

Obviously, these "set" and "jump" instructions decrease performance, because some processors are left idle. In order to improve performance efficiency of PULSE, a codesign solution based on processor/coprocessor model [PRINC], composed of PULSE chips (calculation engine as hardware) and a TMS320C40 processor from Texas Instruments [TEX90] (for management and processing of segmental code), has been experimented.

6.1 System Architecture Composed of one PULSE Chip and a C40

The TMS320C40 is a 32 bit, floating-point processor. Its Central Processing Unit (CPU) is configured for high-speed internal parallelism for the highest sustained performance. It contains a 40/32-bit floating-point/integer unit that supports multiply, divide, square-root and arithmetic logic operations. The C40 has six on-chip communication ports (20M-byte/s bidirectional interface and separate 8-word-deep input/output FIFO for processor to processor communication with no external hardware and simple communication software).

As shown in Figure 6-1, the image processing C40/PULSE system is composed of seven main functional blocks : a TMS320C40 (C40) as coprocessor, a PULSE chip as processor, a glue logic unit realized with FPGA technology, three memories and one oscillator (OSC). The C40 is a common DSP processor that can compensate the weakness of the PULSE chip. The C40 has three main functions, (1) transfers data from global memory to local memory; (2) output all calculation results from PULSE and itself to

global memory; (3) run accumulation and "if" insrtuctions. PULSE gets data from local memory to run parallel code with no or very little 'if' instructions. The 32 bits DRAM is used as a buffer to store input data, as well as partial and final results. The 66 bits DRAM is a program memory or data memory. The glue logic unit (FPGA) makes the system easier to modify.

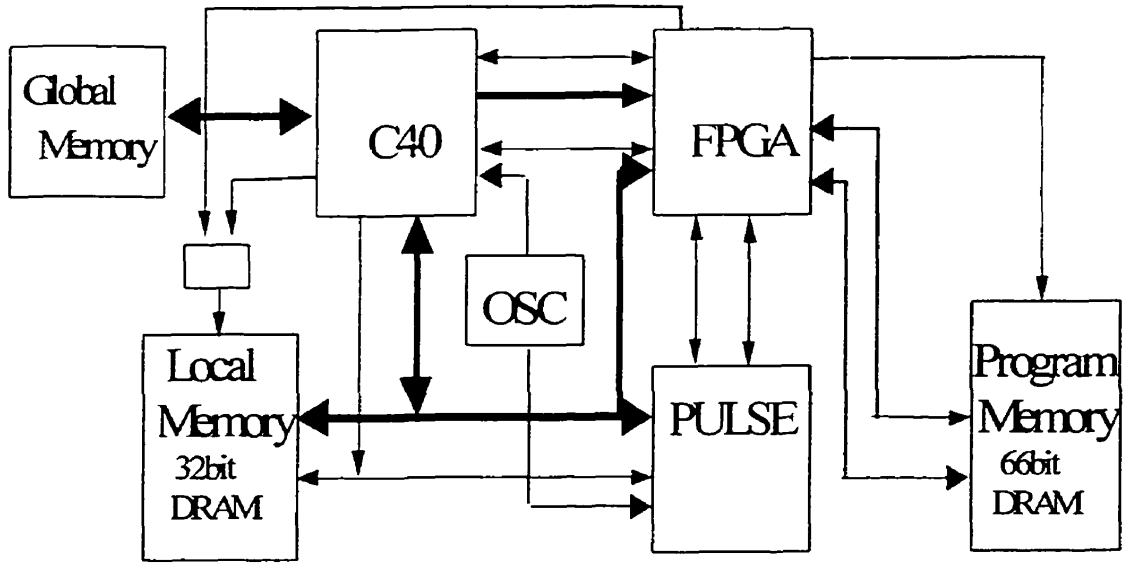


Figure 6-1 The C40/PULSE system

The intermediate result of motion estimation (before accumulate) are sent to local memory. The accumulation and "If" instructions are processed by the C40. Figure 6-2 shows a flowchart for a program split for the proposed heterogeneous architecture. It suggests that the PULSE chip runs "sub"and "abs" instruction and the C40 coprocessor runs "accumulate" and "if" instructions. Here, the major problem is the synchronization between the C40 and the PULSE chip, more precisely data communication. In order to

transfer data promptly, a flag is set in local memory after the 'abs' finished. The C40 checks the flag unit and then start "accumulate" while flag is not zero. An interrupt is generated from PULSE to C40 when PULSE output results to local memory. Of course, the C40's processing time must be less than PULSE's processing time in each search period. In this case, C40 can process interrupts.

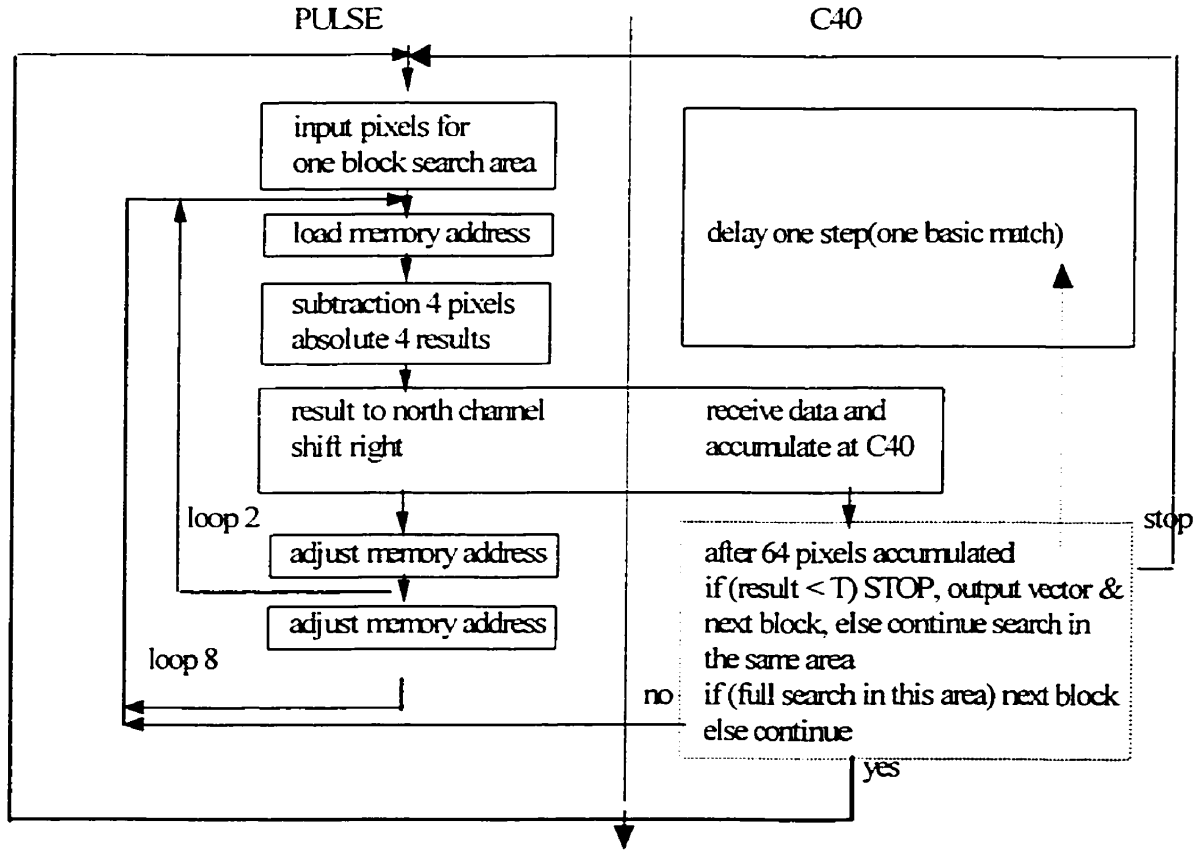


Figure 6-2. Program flowchart for the C40/PULSE system

From the PULSE chip implementation (Figure 4-7). Figure 6-3 presents a possible C40/PULSE system implementation. More precisely, bolded instructions represent instructions executed on the c40 processor. There were about 50% instructions in this part.

```
t1:         ld 0, ra3;          accumulater ra3
            ld rb6, rb1
            push 8;             accumulate 64 pixels
      loop3:
            push 2;             accululate 1 line (8 pixels)
      loop2:
            #2 nop
            ld ra16, addra
            ld rb16, addrb
            #3 nop
            sub *addra, *addrb, ra2
            #3 nop
            abs ra2, ra2;
            #3 nop
            ld ra2, sport
            #2 nop
            [illegible]
      [illegible]
            [illegible]
            [illegible]
            [illegible]
            [illegible]
```

```
[illegible]
add ra16, 4, ra16;        +4
add rb16, 4, rb16;        +4
dbr loop2
add ra16, 8, ra16;        16-4x2=8
add rb16, 8, rb16;        16-4x2=8
dbr loop3
sub ra16, 128, ra16;      -128 back to start point
sub rb16, 128, rb16;      -128 back to start point
ret
```

Figure 6-3 A C40/PULSE system implementation

The PULSE data input time is not included in the program of figure 6-3. Considering GFSM or FSM algorithm, experimentation shows that the C40/PULSE system is about 0.4 times faster than the use of only one PULSE chip.

Data transfer programs for C40 and PULSE are listed in appendix N. The C program and PULSE assembly program listed in Appendix N were simultaneously run in Mentor Graphics simulator (QuickHDL) [PUL96][TEX90].

6.2 Improvement of the C40/PULSE system

In order to increase performance, four PULSE chips are used in this system. This improved system is named C40/4PULSE system and it is presented in Figure 6-4. The four PULSE chips are connected directly into a chain. It is easier to transmit data from

PEs of left PULSE chip to PEs of right PULSE chip on this chain. A common instruction controls all PEs on the four PULSE chips.

Due to the increase of PE chain size, the loop limits are decreased and the time to inout date is also increased. So, the speed increase is not linear. This system runs 3.5 times faster than using only one PULSE chip and 2.2 times faster than the C40/PULSE system.

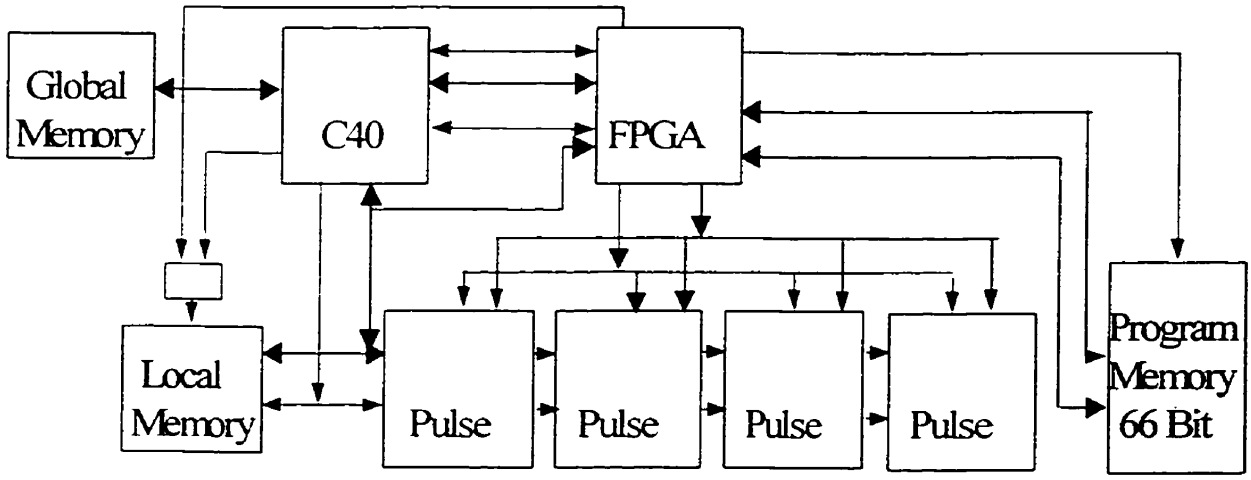


Figure 6-4 The C40/4PULSE system

Table 6.1 compares the processing time of various motion detection algorithms for three architectures: one PULSE chip, the C40/PULSE system (one C40 and one PULSE chip) and the C40/4PULSE system (one C40 and four PULSE chips). The comparison is assumes on image size of 760x480 pixels, The search block is 8x8 pixels and the search area is 16x16 pixels.

According to the different algorithms, the number of search steps varies from 11 to 81 (see table 6-1). In the C40/PULSE system, the processors are working in parallel and the C40 processing time is always less than PULSE processing time. The maximum time is obtained for the one PULSE chip system. So, the Block Match Time (BMT) with only one PULSE chip is 335 instruction cycles. While it 78 instruction cycles are required for the C40/4PULSE system.

The FSM algorithm is slow but accurate. This explains its popularity. GFSM is a bit faster than FSM based on table 6.1. Although GFSM is more complex, in practice, the change between an old picture and new picture is often limited at rate of 30 frames per second. Thus, with GFSM, the number of search steps is generally much lower than the maximum value (81 in this case).

	Maximum Search step	one PULSE chip	the C40/PULSE system (one C40 and one PULSE chip)	the C40/4PULSE system (one C40 and four PULSE chips)
GFSM	81	355 inst. / BMU 449 inst. / pixel 3.4S / frame	256 inst. / BMU 323 inst. / pixel 2.2S / frame	78 inst. / BMU 98 inst. / pixel 660mS / frame
FSM	81	355 inst. / BMU 450 inst. / pixel 3.07S / frame	256 inst. / BMU 360 inst. / pixel 2.4S / frame	78 inst. / BMU 109 inst. / pixel 744mS / frame
CDS	11	355 inst. / BMU 66.5 inst. / pixel 453mS / frame	256 inst. / BMU 42.8 inst. / pixel 292mS / frame	78 inst. / BMU 14.2 inst. / pixel 97mS / frame
3 step	25	355 inst. / BMU 144 inst. / pixel 982mS / frame	256 inst. / BMU 97 inst. / pixel 663mS / frame	78 inst. / BMU 32.3 inst. / pixel 221mS / frame
CSA	13	355 inst. / BMU 75 inst. / pixel 511mS / frame	256 inst. / BMU 50 inst. / pixel 343mS / frame	78 inst. / BMU 16.8 inst. / pixel 114mS / frame

*BMU = Basic Match Unit S = Second inst. = instruction

Table 6-1 Comparison searching results with different algorithms

CHAPTER 7

Conclusions

7.1 Results

This thesis highlights and explains the key features of how a fixed-point SIMD array processor, PULSE, can be used to implement MPEG-2 standard codec that performs video compression. In particular, this research showed effective methods of partitioning algorithms to exploit the parallelism of the PULSE processors.

The main components of an MPEG-2 codec, the motion estimation that computes motion vectors, was designed and simulated. Its performance was evaluated and the algorithms were optimized for the PULSE architecture. It was found that low complexity motion estimation algorithms are faster but often inaccurate (See 2.2). The GFSM algorithm is a better method than the other algorithms, because it is accurate but nevertheless significantly faster than the FSM. Finally, we here shown that 68 chips are required to achieve the motion estimation under the MPEG standard.

Another time consuming operation of the MPEG-2 standard used to reduce the spatial redundancy is the Discrete Cosine Transform (DCT) and its inverse (IDCT). An implementation of the DCT exploiting the parallelism of the PULSE chip and requiring

19 chips has been presented. An important point to underline in the implementation is the use of a predefined cosine table to perform the DCT. The use of this cosine table, first computed on a workstation and afterwards loaded on the PE memories, reduces the processing time, and also increases the accuracy of calculation (due to the precision offered by a language like C++ on a workstation, compared to the 16 bit PULSE processor).

Finally, we have shown that the convolution program is a good application for the PULSE chip applied to image processing. In particular, techniques that reduce the running time of loops and increases the efficiency of performance have been presented (Chapter 3). We should that only 3.75 instructions are required on average to process each pixel with one PULSE chip.

7.2 Future work

Some issues are worth considering for further research. These issues are as follows:

Parallel Architecture: It was proved that the architecture of the PULSE chain affects the performance of MPEG-2 codec. Using a PULSE chain of excessive length is not effective. This could lead to propose a new architecture for the PULSE chain. The array could be divided into several macro blocks, where short PULSE chains process a macro

block. In order to support these parallel PULSE chains, multiple local memories or multi-port memories needed. The data transfer from global memory to local memories are expected to become a bottleneck. For this reason, several C40s could be used. One of them for DMA control and others could be used to process data. In this system, a PULSE chain could have different lengths for processing different algorithms. For example, eight parallel PULSE chains, each comprising four PULSE chips, could support motion estimation, and other chains with only two PULSE chips could support DCT and IDCT.

Management: For each macro block, its processing time could be different. For instance, if all PULSE chains have only one control unit, then the system must wait the slowest one done to continue processing. By contrast, if each chain has its own control unit, each chain can process at its own pace. For implementing this kind of architecture, multiple program memories are required to supply multiple instruction streams.

Program: In the view of appendix F "Program of motion estimation" and appendix H "DCT program of PULSE", it is known that motion estimation and DCT programs are not as efficient as the convolution program, due to some waiting instructions in the program (see 5.2.1 and Appendix H). Generally speaking, these wait cycles appear after set address instructions and calculation instructions. For the first situation, adjusted input data sequence may reduce address change, thus fewer address setting instructions are

required. For the second situation, input instructions should be inserted as much possible as in the waiting times.

Furthermore, as shown in figure 2-3, the MPEG-2 codec includes DCT and IDCT algorithms. For an 8 by 8 DCT, only 64 PE memory units are used to store cosine table and 64 PE memory units are used to store data. The other 128 PE memory units can be used to store DCT result. This DCT result can be used as the input of IDCT. Using this idea will reduce data input time for IDCT.

Others: Using FPGA technology for logic control can make PULSE chips easier to connect with PCI or other standard interfaces for practical applications.

References

1. [16092] Peter A. Ruetz, Po Tong, "A 160-Mpixel/s IDCT Processor for HDTV" IEEE Micro 1992

2. [APP98] Roger Woods, David Trainor, Jean-Paul Heron, Queen's University of Belfast "Applying an XC6200 to Real-Time Image Processing" IEEE Design & Test of Computers, January-March 1998

3. [ARC92] Obed Duardo, Scott C. Knauer, John N. Mailhot, Kalyan Mondal, Tommy C. Poon, "Architecture and Implementation of Ics for a DSC-HDTV Video Decoder System" IEEE Micro 1992

4. [CRO90] M. Ghanbari, "The Cross-Search Algorithm for Motion Estimation" IEEE Transactions on Communication, Vol. 38, No. 7, July 1990

5. [DIG94] Robert Hopkins, ATSC Washington, DC "Digital Terrestrial HDTV for North America: The Grand Alliance HDTV System" IEEE Transactions on Consumer Electronics, Vol. 40, No. 3, August 1994

6. [DIS81] Jaswant R. Jain, Anil K. Jain "Displacement Measurement and Its Application in Interframe Image Coding" IEEE Transactions on Communications, Vol. Com-29, No. 12, December 1981

7. [EFF91] Liang-Gee Chen, Wai-Ting Chen, Yeu-Shen Jehng, and Tzi-Dar Chiueh "An Efficient Parallel Motion Estimation Algorithm for Digital Image Processing" IEEE Transactions on Circuits and Systems for Video Technology, Vol. 1, No. 4, December 1991

8. [FAS87] Hsieh S. Hou, "A Fast Recursive Algorithm For Computing the Discrete Cosine Transform" IEEE Transactions on Acoustics, Speech, And signal Processing, Vol. Assp-35, No. 10, October 1987

9. [GUE92] Gilles Privat, Eric Petajan, "Guest Editors' Introduction Processing Hardware for Real-Time Video Coding" IEEE Micro, October 1992

10. [HAR96] Mitsuo Ikeda, Tsuneo Okubo, Tetsuya Abe, Yoshinori Ito, Yutaka Tashiro and Ryota Kasai "A Hardware/Software Concurrent Design for a Real-Time SP@ML MPEG2 Video-Encoder Chip Set" 1006-1409/96 1996 IEEE

11. [HAR97] Paul Kalapathy, "Hardware-Software Interactions on Mpact" IEEE Micro March/April 1997

12. [HIG93] Laurent Letellier, Didier Juvin, "High Performance Graphics on a SIMD Linear Processor Array" 0-7803-1254-6/93 1993 IEEE

13. [HIG95] Chen-Mie Wu, Dah-Jyh Perng, Wen-Tsung Cheng and Jian-Shing Ho, Department of Electronic Engineering National Taiwan Institute of Technology Taipei, Taiwan, R.O.C. "A High-performance System for Real-Time Video Image Compression Applications" IEEE Transactions on Consumer Electronics, Vol. 41, No. 1, Feruary 1995

14. [IEE90] "IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform" Sponsor CAS Standards Committee of the IEEE Circuits and Systems Society, Approved December 6, 1990, IEEE Standards Board

15. [INT95] Edward R. Dorgherty. Phillip A. Loplante "Entroduction to Real Time Image"

16. [MED94] Woobin Lee and Yongmin Kim. University of Washington. Robert J. Gove and Christopher J. Read. Texas Instruments. "Media Station 5000: Integrating Video and Audio" IEEE Multimedia Summer 1994

17. [MPG97] Joan L. Mitchell. William B. Pennebaker. Chad E. fogg. and Didier J. LeGall. "MPEG Video Compression Standard" International Thomsonp Publishing. 1997

18. [MUL94] Borko furht. Florida Atlantic University. "Multimedia systems: An Overview" IEEE MultiMedia 1994. pp.47-59.

19. [PRE85] Ran Srinivasan and K. R. Rao. "Predictive Coding Based on Efficient Motion Estimation" IEEE Transactions on Communications. Vol. Com-33. No. 8. august 1985

20. [PRINC] J. Staunstrup W. Wolf. "Hardware/Software Co_design: Principles and Practice" Kluwer Acadamic Publishers. Chapter 3

21. [PRO92] Doug Bailey. Natthew Cressa. etc.. "Programmable Vision Processor/Controller for Flexible Implementation of Current and Future Image Compression Standards" IEEE Micro. October 1992

22. [PUL96] PULSE Development team. "PULSE Technical Report Composite Decument" École Polytechnique de Montréal. 1996

23. [REC97] Alexander Bugeja and Woodward Yang "A Reconfigurable VLSI Coprocessing System for the Block Matching Algorithm" IEEE

Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 5, No. 3, September 1997

24. [ROL94] Bryan Ackland, "The Role of VLSI in Multimedia" IEEE Journal of Solid-State Circuits, Vol. 29, No. 4 April 1994

25. [TEL82] Tatsuo Ishiguro and Kazumoto Iinuma, "Television Bandwidth Compression Transmission by Motion-Compensated Interframe Coding" IEEE Communications Magazine, Movember 1982

26. [TEX90] Texas Instruments, "MTS320C40 user's book"

27. [TWO96] Toshio Kondo, Kazuhito Suguri, etc., "Two-Chip MPEG-2 Video Encoder" IEEE Micro April 1996

28. [VID97] William Chien, "Video for Everyone" January 1997 BYTE

29. [VLS96] Cesar Sanz, Matias J. Garrido, Juan M. Meneses "VLSI Architecture for Motion Estimation using the Block-Matching Algorithm" IEEE

Appendix

Appendix A PULSE V1 Competitive analysis

Features	CNAPS	SHARC	TI C80	Oxford A236	PULSE v1
Architecture	SIMD	Single-processor	MIMD	SIMD	SIMD
Clock Frequency	20~25 MHz	33~40MHz	50MHz	40MHz	54MHz
Number of Processor on the chip	64 PNs. Without Controller	Single floating point Processor	One floating-point Processor	One Controller 4 fixed-point processors	One Controller 4 fixed-point processors
Inter-PE Communication support	Very Weak. 5 Mbytes/s	N/A	Cross Bar memory access	No inter-PE communication	Strong. Multichannel 432 Mbytes/s
Parallel Operations in the PE	Very weak Multiply-acc	Strong Two adders one multiplier	Strong 3-input ALU Multiply-acc	Weak Multiply-acc multiply-add	Very strong 3-input, 3-output ALU, Mult-add-acc add-acc

					med-add
Non-linear processing	No	Weak Only max, min and clip of two data	Weak	No	Very strong Max, Min, Med Rank-order Index ranking Core function chip
Application Mapping	Very restricted	.....	Very flexible, But very hard to program	Very restricted	Flexible and easy to program
Scalability	Scalable	Scalable	No	Scalable	Scalable
External Memory and I/O Interface	No memory interface, 8-bit I/O	4 buses, 64-bit datapath to SRAM 10 DMA Channels, 160	Single 64-bit Bus shared by all the processors for data and program	400 Mbytes/s 32-bit sync. Memory 2 40 Mbytes/s	Two buses 432 Mb/s sync. Memory 4 108 Mb/s data ports

		Mbytes/s		DMA ports	
On-chip memory	4kbytes on each Pn	2Mbits or 4Mbits	50Kbytes	1kbytes Ins. 1kbytes data	2kbytes Ins. 2.5kbytes data
Micro-Instruction	64-bit total 32-bit control 32-bit PN	48-bit	64-bit for parallel proc. 32-bit master	32-bit	64-bit parallel
Instruction-set	Very limited	Rich, extended for non-linear processing	Rich, extended for logic processing	Very limited	Very rich extended for both linear and non-linear proc.
Software Tools	Assembler C compiler Debugger	Assembler C compiler Simulator Debugger Evaluation board	Assembler C compiler Simulator Debugger Evaluation board	Software development kit	Assembler C compiler Simulator Debugger Evaluation board Application libraries

Packaging	200-pin PN 240-pin CSC PGA	240-pin PQFP	305-pin ceramic PGA	208-pin PQFP	240-pin PQFP
Availability	Yes	Yes	Yes	Q2 1996	Q4 1996
Cost	High	High	High	Low	Low

Appendix B PULSE V1 technical features

The following list is a summary of the features of PULSE V1:

4 processing elements (PEs) per chip

54 MHz operation (worst case)

216 Mega MACCs per second (16x16 MACC)

216 Mega 3-operand ALU ops per second

216 Mega 32-bit shift/rotate per second

4-stage execution pipeline

ES2 ECPD07 0.6 um process

Six modulo counters for easy modulo addressing

2 16-bit shift-register chains for inter-processor communications and data I/O

32-bit accumulate chain for direct data link between neighbor PEs

Instruction set includes vector and parallel instructions

Specialized 3 operand arithmetic-logic unit

Single cycle rank, clip, cor, max, med and min operations on 3 course operands

Accumulator has 33-bit internal range

Programmable overflow saturation for both signed and unsigned values

Signed saturates to +2 –1 and –2

Unsigned saturates to 0 and +2 –1

Four 16-bit reconfigurable data ports 432Mb/s of data I/O

Synchronous, asynchronous pseudo-synchronous

Up to 108 Mbytes/second per port

2 ports may be configured as C40 type COM ports

256 word internal program memory

64 bit external program data bus

May be used as 432 Mb/s data port

256 word internal constants memory

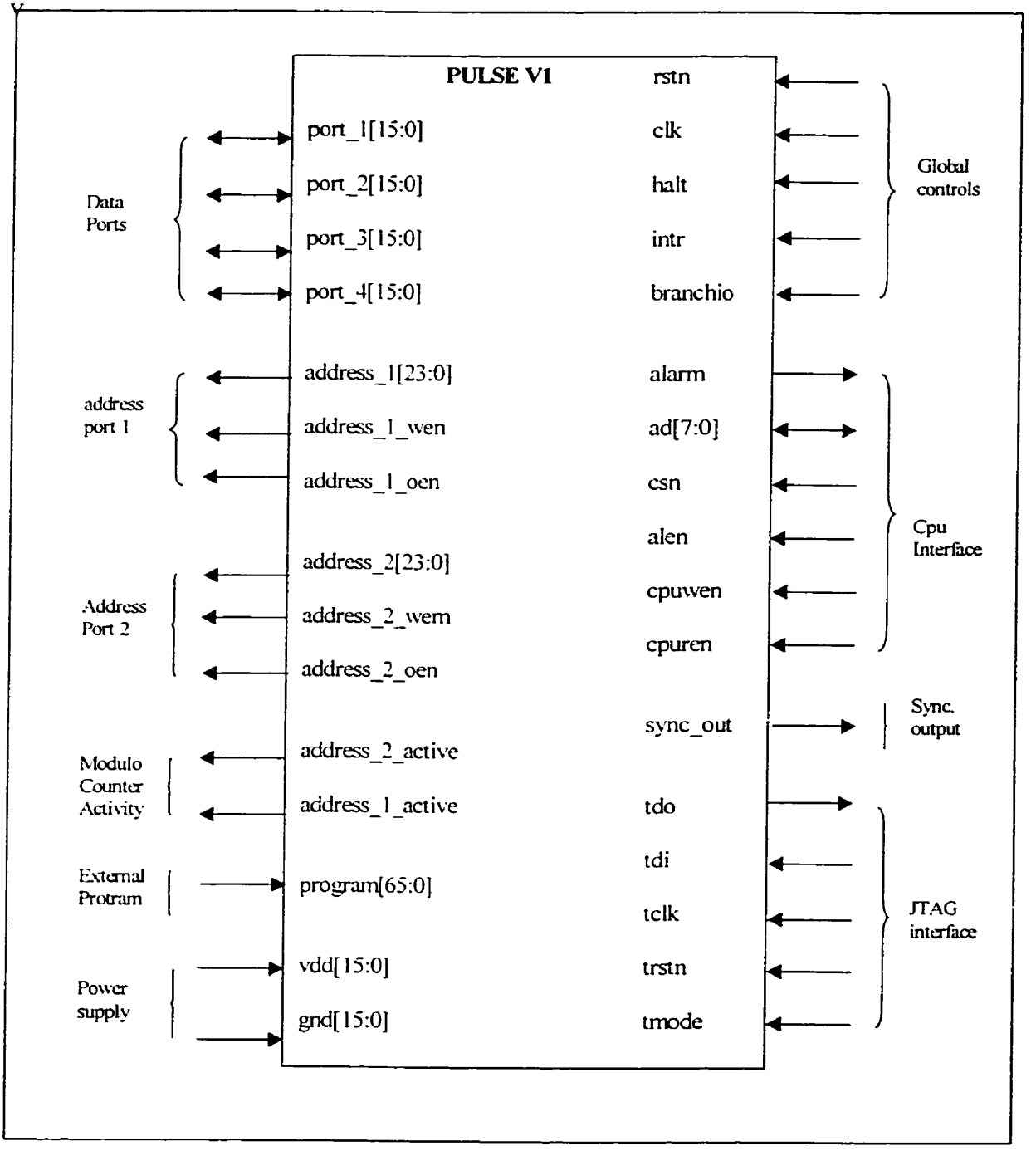
Used for coefficient storage for filter algorithms etc.

Two 24 bit flexible address ports

- Programmable modulo counters available for address generation

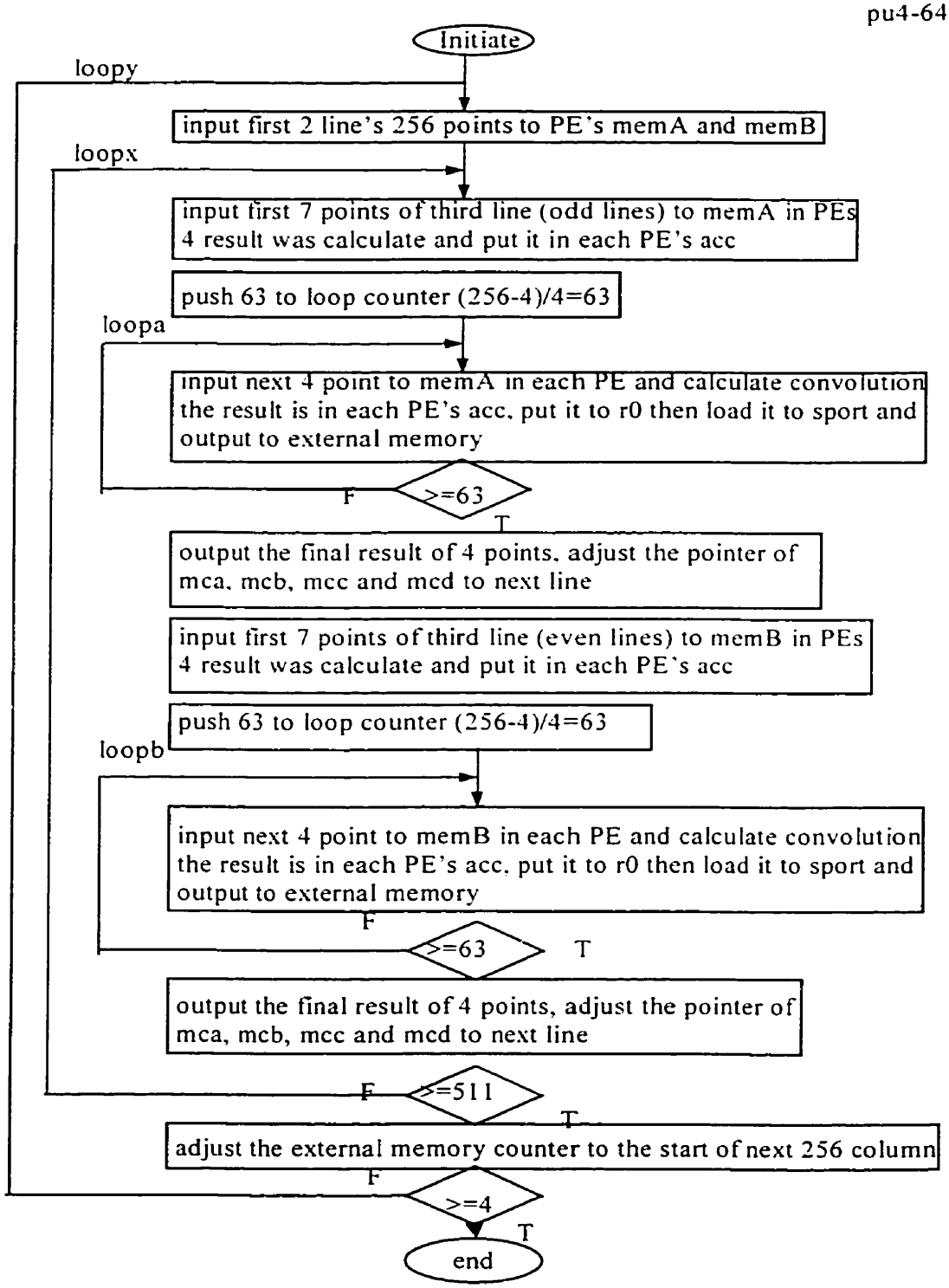
- Standard CPU interface for configuration and status
  - Configuration can also be performed by program
- PULSE chips are cascadable to form larger arrays of processors
  - Synchronous I/O ports allow direct connection between chips to form large linear arrays of processors
  - Two dimensional arrays are possible since there are 4 I/O ports

Appendix C PULSE V1 logic symbol-subject to design review



Appendix D The convolution program flowchart and program

pu4-64



```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PULSE  CONVOLUTION.ASM (3x3 sample window,  1kx1k pixel ;
;image) pu4-6                                              ;
;                              assemble syntax passed      ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
            .init
            .ra11 1, 1, 1, 1, 1, 1, 1, 1, 1

            .text
            ldeamc 0, 0, mc_min, mc_min
            ldeamc 1048576, 1048576, mc_max, mc_max      ;(1024*1024)
            ldeamc 1, 1, mc_stride, mc_stride
            ldeamc 0, 1025, mc_start, mc_start  ;start from second line and second point(1024+1)
            ldiamc 3, 0, 255, mcar          ;read from 4th point
            ldiamc 0, 0, 255, mcaw
            ldiamc 3, 0, 255, mcbr          ;read from 4th point
            ldiamc 0, 0, 255, mcbw

            push 4                  ;(col=4)
loopy:      push 257

loopm:      fwd nport, *mcaw(1) || nsr || io *mccr%     ;start input 256 point of first line
            dbr loopm
            ldeamc  766, 1, mc_stride, mc_stride         ;(+1024-256-2)
            fwd nport, *mcaw(1) || nsr || io *mccr%     ;end input 256+2 point of first line
            ldeamc  1, 1, mc_stride, mc_stride

            push 257
loopn:      fwd nport, *mcbw(1) || nsr || io     *mccr% ;start input 256 point of second line
            dbr loopn
            ldeamc  766, 1, mc_stride, mc_stride         ;(+1024-256-2)
            fwd nport, *mcbw(1) || nsr || io *mccr%     ;end input 256+2 point of second line
            ldeamc 1, 1, mc_stride, mc_stride


;THIRD LINE        (mcarw=0, mcbrw=0, mccr=0 at 3rd line, mcdw=1 at 2nd  line)

            push 511                        ;((1024-2)/2)
loopx:                                      ;(next line)
;*********
            #3 fwd nport, *mcaw(1) || nsr || io *mccr%  ;(input 1~3p and point to 4th)
            mult *mcar(1), ra11, acc
            fwd nport, *mcaw(1) || nsr || io *mccr%
            mult *mcar(1), ra12, acc+
            fwd nport, *mcaw(1) || nsr || io *mccr%
            mult *mcar(-2), ra13, acc+
            mult *mcbr(1), ra14, acc+
            fwd nport, *mcaw(2) || nsr || io *mccr%
            mult *mcbr(1), ra15, acc+ || nsr || io *mccr%
            mult *mcbr(2), ra16, acc+
            mult *mcar(1), ra17, acc+
            mult *mcar(1), ra18, acc+
```

```
                mult *mcar(2), ra19, acc+
        ;already input 7 point. The result of first 4 point is at acc

                push 63                 :(256-4)/4=63
loopa:          mult *mcar(1), ra11, acc
                fwd nport, *mcaw(1) || nsr || io *mccr%
                madd ra1, 0, acc, r0
                mult *mcar(1), ra12, acc+
                fwd nport, *mcaw(1) || nsr || io *mccr%
                mult *mcar(-2), ra13, acc+
                ld rb0, sport
                mult *mcbr(1), ra14, acc+
                fwd nport, *mcaw(2) || nsr || io *mccr%
                mult *mcbr(1), ra15, acc+ || nsr || io *mccr%
                mult *mcbr(2), ra16, acc+ || ssr || io *mcdw%
                mult *mcar(1), ra17, acc+ || ssr || io *mcdw%
                mult *mcar(1), ra18, acc+ || ssr || io *mcdw%
                mult *mcar(2), ra19, acc+ || ssr || io *mcdw%
                dbr loopa

                #2 nop
                madd ra1, 0, acc, r0
                #3 nop
                ld rb0, sport
                #3 nop
                #3 ssr || io *mcdw%
                ldeamc 766, 766, mc_stride, mc_stride       :!!! (1024-(256+2))
                ssr || io *mcdw%, *mccr%

                ldeamc 1, 1, mc_stride, mc_stride
                                        :(mccr=new line 1st point, mcdw=new line 2nd  point)
                ldiamc 0, 0, 255, mcaw
                ldiamc 3, 0, 255, mcbr
                ldiamc 3, 0, 255, mcar
                ldiamc 0, 0, 255, mcbw          :(mcaw, mcbw=0, mcar, mcbr=3 the 4th point)
;*********
;****************************************
                #3 fwd nport, *mcbw(1) || nsr || io *mccr%  :(input 1~3p and point to 4th)
                mult *mcbr(1), ra11, acc
                fwd nport, *mcbw(1) || nsr || io *mccr%
                mult *mcbr(1), ra12, acc+
                fwd nport, *mcbw(1) || nsr || io *mccr%
                mult *mcbr(-2), ra13, acc+
                mult *mcar(1), ra14, acc+
                fwd nport, *mcbw(2) || nsr || io *mccr%
                mult *mcar(1), ra15, acc+ || nsr || io *mccr%
                mult *mcar(2), ra16, acc+
                mult *mcbr(1), ra17, acc+
                mult *mcbr(1), ra18, acc+
                mult *mcbr(2), ra19, acc+

        ;already input 7 point. The result of first 4 point is at acc
```

```
                push 63                         :(256-4)/4=63
loopb:          mult *mcbr(1), ra11, acc
                fwd nport, *mcbw(1) || nsr || io *mccr%
                madd ra1, 0, acc, r0
                mult *mcbr(1), ra12, acc+
                fwd nport, *mcbw(1) || nsr || io *mccr%
                mult *mcbr(-2), ra13, acc+
                ld rb0, sport
                mult *mcar(1), ra14, acc+
                fwd nport, *mcbw(2) || nsr || io *mccr%
                mult *mcar(1), ra15, acc+ || nsr || io *mccr%
                mult *mcar(2), ra16, acc+ || ssr || io *mcdw%
                mult *mcbr(1), ra17, acc+ || ssr || io *mcdw%
                mult *mcbr(1), ra18, acc+ || ssr || io *mcdw%
                mult *mcbr(2), ra19, acc+ || ssr || io *mcdw%
                dbr loopb

                #2 nop
                madd ra1, 0, acc, r0
                #3 nop
                ld rb0, sport
                #3 nop
                #3 ssr || io *mcdw%
                ldeamc 766, 766, mc_stride, mc_stride       :!!! (1024-(256+2))
                ssr || io *mcdw%, *mccr%

                ldeamc 1, 1, mc_stride, mc_stride
                                        :(mccr=new line 1st point, mcdw=new line 2nd  point)
                ldiamc 0, 0, 255, mcaw
                ldiamc 3, 0, 255, mcbr
                ldiamc 3, 0, 255, mcar
                ldiamc 0, 0, 255, mcbw          :(mcaw, mcbw=0, mcar, mcbr=3 the 4th point)
;*********************************
                dbr loopx                       :(cyc in line)

                ldeamc 256, 2304, mc_stride, mc_stride:(-1024*1024+256,-1022*1024+256)
                io *mccr%, *mcdw%
                ldeamc 1, 1, mc_stride, mc_stride

                dbr loopy
                end

                .end
```

Appendix E Convolution program (3x3 window with 32x32 pixel image), data of source image and data of result image

1)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  PULSE  CONVOLUTION.ASM     (3x3 sample window,  32x32  ;
;                              pixel image) pu97-1        ;
;                                                         ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
              .init
              .ra11  1, 1, 1, 1, 1, 1, 1, 1, 1

              .text
              ldeamc 0, 0, mc_min, mc_min
              ldeamc  1023, 1023, mc_max, mc_max      :(32*32)
              ldeamc 1, 1, mc_stride, mc_stride
              ldeamc 0, 32, mc_start, mc_start  ;start from second line and second point(31+1)
              ldiamc 3, 0, 255, mcar            ;read from 4th point
              ldiamc 0, 0, 255, mcaw
              ldiamc 3, 0, 255, mcbr            ;read from 4th point
              ldiamc 0, 0, 255, mcbw

              push 32
loopm:        fwd nport, *mcaw(1) || nsr || io *mccr%      ;start input 32 points of first line
              dbr loopm
              push 32
loopn:        fwd nport, *mcbw(1) || nsr || io      *mccr% ;start input 32  points of second line
              dbr loopn

;THIRD LINE          (mcarw=0, mcbrw=0, mccr=0 at 3rd line, mcdw=1 at 2nd  line)
              push 15                    :(32-2)/2=15   ((1024-2)/2)
loopx:                                          :(next line)
;*********

              #3 fwd nport, *mcaw(1) || nsr || io *mccr%  :(input 1~3p and point to 4th)
              mult *mcar(1), ra11, acc
              fwd nport, *mcaw(1) || nsr || io *mccr%
              mult *mcar(1), ra12, acc+
              fwd nport, *mcaw(1) || nsr || io *mccr%
              mult *mcar(-2), ra13, acc+
              mult *mcbr(1), ra14, acc+
              fwd nport, *mcaw(2) || nsr || io *mccr%
              mult *mcbr(1), ra15, acc+ || nsr || io *mccr%
              mult *mcbr(2), ra16, acc+
              mult *mcar(1), ra17, acc+
              mult *mcar(1), ra18, acc+
              mult *mcar(2), ra19, acc+
        ;already input 7 point. The result of first 4 point is at acc
              nop||io *mcdw%                  ;pro-set first result point=xx
              push 7                     ;(32-4)/4=7   (256-4)/4=63
loopa:        mult *mcar(1), ra11, acc
```

```
                fwd nport, *mcaw(1) || nsr || io *mccr%
                madd ra1, 0, acc, r0
                mult *mcar(1), ra12, acc+
                fwd nport, *mcaw(1) || nsr || io *mccr%
                mult *mcar(-2), ra13, acc+
                ld rb0, sport
                mult *mcbr(1), ra14, acc+
                fwd nport, *mcaw(2) || nsr || io *mccr%
                mult *mcbr(1), ra15, acc+ || nsr || io *mccr%
                mult *mcbr(2), ra16, acc+ || ssr || io *mcdw%
                mult *mcar(1), ra17, acc+ || ssr || io *mcdw%
                mult *mcar(1), ra18, acc+ || ssr || io *mcdw%
                mult *mcar(2), ra19, acc+ || ssr || io *mcdw%
                dbr loopa

;*********
;*****************************************
                #3 fwd nport, *mcbw(1) || nsr || io *mccr%  ;(input 1~3p and point to 4th)
                mult *mcbr(1), ra11, acc
                fwd nport, *mcbw(1) || nsr || io *mccr%
                mult *mcbr(1), ra12, acc+
                fwd nport, *mcbw(1) || nsr || io *mccr%
                mult *mcbr(-2), ra13, acc+
                mult *mcar(1), ra14, acc+
                fwd nport, *mcbw(2) || nsr || io *mccr%
                mult *mcar(1), ra15, acc+ || nsr || io *mccr%
                mult *mcar(2), ra16, acc+
                mult *mcbr(1), ra17, acc+
                mult *mcbr(1), ra18, acc+
                mult *mcbr(2), ra19, acc+

        ;already input 7 point. The result of first 4 point is at acc
                nop||io *mcdw%
                push 7                          ;(32-4)/4=7
loopb:          mult *mcbr(1), ra11, acc
                fwd nport, *mcbw(1) || nsr || io *mccr%
                madd ra1, 0, acc, r0
                mult *mcbr(1), ra12, acc+
                fwd nport, *mcbw(1) || nsr || io *mccr%
                mult *mcbr(-2), ra13, acc+
                ld rb0, sport
                mult *mcar(1), ra14, acc+
                fwd nport, *mcbw(2) || nsr || io *mccr%
                mult *mcar(1), ra15, acc+ || nsr || io *mccr%
                mult *mcar(2), ra16, acc+ || ssr || io *mcdw%
                mult *mcbr(1), ra17, acc+ || ssr || io *mcdw%
                mult *mcbr(1), ra18, acc+ || ssr || io *mcdw%
                mult *mcbr(2), ra19, acc+ || ssr || io *mcdw%
                dbr loopb
;*********************************
                dbr loopx                               ;(cyc in line)
                end
                .end
```

2) Data of source image (32 x 32 pixel)

```
P2
32 32
255
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  64  64  2  2  2  2  2  2
2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  2  2  2  64  64  64  64  2  2  2  2  2
2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  2  2  64  64  100  100  64  64  2  2
2  2  2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  2  64  64  100  100  100  100  64  64
2  2  2  2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  64  64  100  100  130  130  100  100
64  64  2  2  2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  64  64  100  100  130  130  130  130  100
100  64  64  2  2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  64  64  100  100  130  130  160  160  130
130  100  100  64  64  2  1  1  1  1  1  1  1  1
0  1  1  1  1  1  1  1  64  64  100  100  130  130  160  160  160  160
130  130  100  100  64  64  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  64  64  100  100  130  130  160  160  180  180
160  160  130  130  100  100  64  64  1  1  1  1  1  1  1
1  1  1  1  1  1  64  64  100  100  130  130  160  160  180  180  180
180  160  160  130  130  100  100  64  64  1  1  1  1  1  1
1  1  1  1  1  64  64  100  100  130  130  160  160  180  260  200  200
180  180  160  160  130  130  100  100  64  64  1  1  1  1  1
1  1  1  1  64  64  100  100  130  130  160  160  180  180  200  200
200  200  180  180  160  160  130  130  100  100  64  64  1  1  1  1
1  1  1  64  64  100  100  130  130  160  160  180  180  200  200  220
220  200  200  180  180  160  160  130  130  100  100  64  64  1  1  1
1  1  64  64  100  100  130  130  160  160  180  180  200  200  220
220  220  220  200  200  180  180  160  160  130  130  100  100  64  64
1  1
1  64  64  100  100  130  130  160  160  180  180  200  200  220  220
250  250  220  220  200  200  180  180  160  160  130  130  100  100
64  64  1
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  64  64  2  2  2  2  2  2
2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  2  2  2  64  64  64  64  2  2  2  2  2
2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  2  2  64  64  100  100  64  64  2  2
2  2  2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  2  64  64  100  100  100  100  64  64
2  2  2  2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  2  64  64  100  100  130  130  100  100
64  64  2  2  2  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  2  2  64  64  100  100  130  130  130  130  100
100  64  64  2  2  1  1  1  1  1  1  1  1
```

```
1  1  1  1  1  1  1  1  2  64  64  100  100  130  130  160  160  130
130  100  100  64  64  2  1  1  1  1  1  1  1  1
0  1  1  1  1  1  1  1  64  64  100  100  130  130  160  160  160  160
130  130  100  100  64  64  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  64  64  100  100  130  130  160  160  180  180
160  160  130  130  100  100  64  64  1  1  1  1  1  1  1
1  1  1  1  1  1  64  64  100  100  130  130  160  160  180  180  180
180  160  160  130  130  100  100  64  64  1  1  1  1  1  1
1  1  1  1  1  64  64  100  100  130  130  160  160  180  180  200  200
180  180  160  160  130  130  100  100  64  64  1  1  1  1  1
1  1  1  1  64  64  100  100  130  130  160  160  180  180  200  200
200  200  180  180  160  160  130  130  100  100  64  64  1  1  1  1
1  1  1  64  64  100  100  130  130  160  160  180  180  200  200  220
220  200  200  180  180  160  160  130  130  100  100  64  64  1  1  1
1  1  64  64  100  100  130  130  160  160  180  180  200  200  220
220  220  220  200  200  180  180  160  160  130  130  100  100  64  64
1  1
1 64 64 100 100 130 130 160 160 180 180 200 200 220 220 250 250 220 220 200 200 180
180 160 160 130 130 100 100 64 64 1
```

3) Data of result image (32 x 32 pixel)

```
P2
32 32
255
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 1 1 1 1 1 16 32 32 24 9 1 1 1 1 1 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 2 2 2 2 17 40 60 65 45 25 9 2 2 2 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 2 2 2 17 40 65 82 82 69 45 25 9 2 2 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 2 2 17 40 65 86 99 103 90 69 45 25 9 2 1 1 1 1 1 1 1 1
1 1
1 1 1 1 1 1 1 1 2 17 40 65 86 103 115 115 106 90 69 45 25 9 1 1 1 1 1 1
1 1 1 1
1 1 1 1 1 1 1 1 17 40 65 86 103 118 130 133 122 106 90 69 45 25 9 1 1 1
1 1 1 1 1 0
0 1 1 1 1 1 1 17 40 65 86 103 118 133 145 145 137 122 106 90 69 45 25 9
1 1 1 1 1 1 0 1
0 1 1 1 1 1 16 40 65 86 103 118 133 148 158 161 151 137 122 106 90 69
45 24 8 1 1 1 1 1 0 1
0 1 1 1 1 16 40 65 86 103 118 133 148 161 170 170 163 151 137 122 106
90 69 44 24 8 1 1 1 1 1 1
1 1 1 1 16 40 65 86 103 118 133 148 161 182 190 182 175 163 151 137 122
106 90 69 44 24 8 1 1 1 1 1
1 1 1 16 40 65 86 103 118 133 148 161 182 192 200 190 185 175 163 151
137 122 106 90 69 44 24 8 1 1 1 1
1 1 16 40 65 86 103 118 133 148 161 172 192 202 210 202 195 185 175 163
151 137 122 106 90 69 44 24 8 1 1 1
1 16 40 65 86 103 118 133 148 161 172 182 192 202 210 210 205 195 185
175 163 151 137 122 106 90 69 44 24 8 1 1
16 40 65 86 103 118 133 148 161 172 182 192 202 212 221 225 216 205 195
185 175 163 151 137 122 106 90 69 44 24 8 0
```

```
24 44 61 74 86 97 108 118 127 135 142 150 157 166 172 172 166 157 150
142 135 127 118 108 97 86 74 61 44 24 8 1
16 28 33 41 45 52 56 62 65 70 73 78 80 86 98 106 94 80 78 73 70 65 63
56 52 45 41 33 28 16 8 1
0 0 0 0 0 0 0 1 1 1 1 1 1 16 32 32 24 9 1 1 1 1 1 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 2 2 2 2 17 40 60 65 45 25 9 2 2 2 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 2 2 17 40 65 82 82 69 45 25 9 2 2 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 2 2 17 40 65 86 99 103 90 69 45 25 9 2 1 1 1 1 1 1 1 1
1 1
1 1 1 1 1 1 1 1 2 17 40 65 86 103 115 115 106 90 69 45 25 9 1 1 1 1 1 1
1 1 1 1
1 1 1 1 1 1 1 1 17 40 65 86 103 118 130 133 122 106 90 69 45 25 9 1 1 1
1 1 1 1 1 0
0 1 1 1 1 1 1 17 40 65 86 103 118 133 145 145 137 122 106 90 69 45 25 9
1 1 1 1 1 1 0 1
0 1 1 1 1 1 16 40 65 86 103 118 133 148 158 161 151 137 122 106 90 69
45 24 8 1 1 1 1 1 0 1
0 1 1 1 1 16 40 65 86 103 118 133 148 161 170 170 163 151 137 122 106
90 69 44 24 8 1 1 1 1 1 1
1 1 1 1 16 40 65 86 103 118 133 148 161 172 180 182 175 163 151 137 122
106 90 69 44 24 8 1 1 1 1 1
1 1 1 16 40 65 86 103 118 133 148 161 172 182 190 190 185 175 163 151
137 122 106 90 69 44 24 8 1 1 1 1
1 1 16 40 65 86 103 118 133 148 161 172 182 192 200 202 195 185 175 163
151 137 122 106 90 69 44 24 8 1 1 1
1 16 40 65 86 103 118 133 148 161 172 182 192 202 210 210 205 195 185
175 163 151 137 122 106 90 69 44 24 8 1 1
64 64 100 100 130 130 160 160 180 180 200 200 220 220 250 250 220 220
200 200 180 180 160 160 130 130 100 100 64 64 1
16 40 65 86 103 118 133 148 161 172 182 192 202 212 221 225 216 205 195
185 175 163 151 137 122 106 90 69 44 24 8 1
```

Appendix F Program of motion estimation

1) FSM

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     mogcod.asm                                                    ;
;mogcod   PULSE assembler  motion estimation.asm   FSM(8x8block,    ;
;16x16area, 64x64image) call                                        ;
;           frameA is new picture, frameB is old picture            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          .text
     pull:
          ldcr 2, port1config;
                    (90 2) port 1 as input, synchroneous mode
          ldcr 1, port1incon;
                    (A0 1) port 1 connect to north channel (input)
          ldcr 0, port2config;
                    (91 0) port 2 as input
          ldcr 2, port2incon;
                    (A1 2) port 2 connect to south channel (input)
          ldcr 1, port3config
          ldcr 1, port3outcon
          ldcr 3, port4config
          ldcr 2, port4outcon

          ldcr 3, bootcontrol;    address purt1 for modulo counter

          ldeamc 0, 2097152, mc_min, mc_min;
          ldeamc  4095, 2101248, mc_max, mc_max;
                    (64x64+start=4096+2097152)
          ldeamc 0, 2097152, mc_start, mc_start;  2**11=2097152
          ldeamc 1, 1, mc_stride, mc_stride;

;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 1, row 1
          ldiamc 0, 0, 144, mcar; 48+96=144
          ldiamc 0, 0, 144, mcaw; 48+96=144
          ldiamc 0, 0, 144, mcbr; 48+96=144
          ldiamc 0, 0, 144, mcbw; 48+96=144
;$$$$$$$$$$$ input frame A, B 128-64-16=48 pixels
          push 4; 4x12      4 lines
          call t2
          ld 0, ra1;  vector counter number
          ld 0, rb1;  16k, temporal vector address position
;$$$$$$$$$$$ input frame A, B 128-32=96 pixels
          push 8;     8+8
          call t2
;!!!!!!!!col 1, row 1,  calculate
          ld 0, ra16; memA's address
          ld 0, rb16; memB's address
          push 5;     5line loop
          call t4
;!!!!!!!!!first col, first block end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 1, row 1
```

```
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 1, row 2-127
            ldeamc 256, 2097153, mc_start, mc_start;
                                  256(64x4), 2097152+1(vector)
            ldiamc 0, 0, 192, mcar; 96+96=192
            ldiamc 0, 0, 192, mcaw; 96+96=192
            ldiamc 0, 0, 192, mcbr; 96+96=192
            ldiamc 0, 0, 192, mcbw; 96+96=192
            ld 48, ral8;          4x12=48 frameA in memA start point
            ld 0, rb18;           frameB in memB start point
;$$$$$$$$$$$ input frame A, B 128-32=96 pixels
            push 8; 8x12          8 lines
            call t2
            push 6;               column 1 loops(128-2 for boundry)
      loopo:
;$$$$$$$$$$$ input frame A, B 96 pixels
            push 8;               8x12
            call t2
;!!!!!!!!!col 1, 2-127 block calculate
            #2 nop
            ld ral8, ral6;        memA's address
            ld rb18, rb16;        memB's address
            push 9;               9line loop
            call t4
;!!!!!!! One block calculate end
            add ral8, 96, ral8;       8x12
            add rb18, 96, rb18;       8x12
            dbr loopo
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 1, row 2-127
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 1, row 128
;$$$$$$$$$$$ input frame A, B 48 pixels
            push 4;       4
            call t2
;!!!!!!!!!col 1, 128 block calculate
            #2 nop
            ld ral8, ral6;        memA's address
            ld rb18, rb16;        memB's address
            push 5;               9line loop
            call t4
;!!!!!!!!! calculate end
            ldeamc -4092, 1, mc_stride, mc_stride;-4096+(12-4)-3-1=-
4092
;old-4024=-64x(64-1)-8 or +64-8  pointed to next col. Start point
(64x64)
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 1, row 128
;end of col. 1 !!!!!!!!!!!!!!!!!!!!!!!
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 2-127, row 1
            push 6;       126col.s
loopcol:
            ldiamc 0, 0, 192, mcar; 64+128=192
```

```
            ldiamc 0, 0, 192, mcaw; 64+128=192
            ldiamc 0, 0, 192, mcbr; 64+128=192
            ldiamc 0, 0, 192, mcbw; 64+128=192
;$$$$$$$$$$ input frame A, B 16x4=64 pixels
            push 4;           4x16  4 lines
            call t3
;$$$$$$$$$$ input frame A, B 128 pixels
            push 8;           8+8
            call t3
;!!!!!!!!!col 2-127, row 1,  calculate
            ld 4, ra16;       (+4) memA's address
            ld 0, rb16;       memB's address
            push 5;           5line loop
            call t5
;!!!!!!!!!calculate end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 2-127, row 1
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 2-127, row 2-127
            ldeamc -512, 1, mc_stride, mc_stride;     -64x8-3-1
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop;           now is pointed to next line begin
            ldiamc 0, 0, 255, mcar; 128+128=256
            ldiamc 0, 0, 255, mcaw; 128+128=256
            ldiamc 0, 0, 255, mcbr; 128+128=256
            ldiamc 0, 0, 255, mcbw; 128+128=256
            ld 68, ra18;      4x16+4=68 frameA in memA start point
            ld 0, rb18;       frameB in memB start point
;$$$$$$$$$$ input frame A, B 128 pixels
            push 8;           8x16  8 lines
            call t3
            push 6;           column 1 loops(128-2 for boundry)
      loopo1:
;$$$$$$$$$$ input frame A, B 128 pixels
            push 8;           8x16
            call t3
;!!!!!!!!!col 2-127, 2-127 block calculate
            #2 nop
            ld ra18, ra16;    memA's address
            ld rb18, rb16;    memB's address
            push 9;           9line loop
            call t5
;!!!!!!!!! calculate end
            add ra18, 128, ra18;    8x16
            add rb18, 128, rb18;    8x16
            dbr loopo1;
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 2-127, row 2-127
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 2-127, row 128
;$$$$$$$$$$ input frame A, B 64 pixels
            push 4;           4
            call t3
;!!!!!!!!!col 2-127, 128 block calculate
            #2 nop
```

```
            ld ra18, ra16;     memA's address
            ld rb18, rb16;     memB's address
            push 5;            5line loop
            call t5
;!!!!!!!!! calculate end
            ldeamc -4092, 1, mc_stride, mc_stride; -4096+8-3-1=-4092
;old-4024=-64x(64-1)-8 or +64-8  pointed to next col. Start point
(64x64)
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop;            now is pointed to next line begin
            dbr loopcol
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 2-127, row 128
;end of col. 2-127 !!!!!!!!!!!!!!!!!!!!!!!!
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 128, row 1
            ldiamc 0, 0, 144, mcar; 48+96=144
            ldiamc 0, 0, 144, mcaw; 48+96=144
            ldiamc 0, 0, 144, mcbr; 48+96=144
                  ldiamc 0, 0, 144, mcbw; 48+96=144
;$$$$$$$$$$ input frame A, B 12x4=48 pixels
            push 4; 4x12       4 lines
            call t2
;$$$$$$$$$$ input frame A, B 96 pixels
            push 8;            8+8
            call t2
;!!!!!!!!col 128, row 1,  calculate
            ld 4, ra16;        (+4) memA's address
            ld 0, rb16;        memB's address
            push 5;            5line loop
            call t4
;!!!!!!!!! calculate end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 128, row 1
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 128, row 2-127
            ldeamc -512, 1, mc_stride, mc_stride;     -64x8-3-1
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
            ldiamc 0, 0, 192, mcar; 96+96=192
            ldiamc 0, 0, 192, mcaw; 96+96=192
            ldiamc 0, 0, 192, mcbr; 96+96=192
                  ldiamc 0, 0, 192, mcbw; 96+96=192
            ld 52, ra18;       4x12+4=52 frameA in memA start point
            ld 0, rb18;        frameB in memB start point
;$$$$$$$$$$ input frame A, B 128-32=96 pixels
            push 8;            8x12  8 lines
            call t2
            push 6;            column 1 loops(128-2 for boundry)
      loopo2:
;$$$$$$$$$$ input frame A, B 96 pixels
            push 8;            8x12
```

```
            call t2
;!!!!!!!!!col 128, 2-127 block calculate
            #2 nop
            ld ra18, ra16;     memA's address
            ld rb18, rb16;     memB's address
            push 9;            9line loop
            call t4
;!!!!!!!!!! calculate end
            add ra18, 96, ra18;      8x12
            add rb18, 96, rb18;      8x12
            dbr loopc2;
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 128, row 2-127
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 128, row 128
;$$$$$$$$$$$ input frame A, B 48 pixels
            push 4;            4
            call t2
;!!!!!!!!!col 1, 128 block calculate
            #2 nop
            ld ra18, ra16;     memA's address
            ld rb18, rb16;     memB's address
            push 5;            5line loop
            call t4
;!!!!!!!!!! calculate end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 128, row 128
;end of col. 128 !!!!!!!!!!!!!!!!!!!!!!!!
end

t1:         ld 0, ra3;         accumulater ra3
            ld rb6, rb1
            push 8;            accumulate 64 pixels
      loop3:
            push 2;            accululate 1 line (8 pixels)
      loop2:
            #2 nop
            ld ra16, addra
            ld rb16, addrb
            #3 nop
            sub *addra, *addrb, ra2
            #3 nop
            abs ra2, ra2;
            #3 nop
            ld ra2, sport
            #2 nop
            push 4;            accumulate 4 pixels
      loop1:
            ld sport, rb2
            #2 nop
            ssr
            add ra3, rb2, ra3;       accumulater ra3
            dbr loop1
            add ra16, 4, ra16;       +4
            add rb16, 4, rb16;       +4
            dbr loop2
            add ra16, 8, ra16;       16-4x2=8
```

```
            add rb16, 8, rb16;      16-4x2=8
            dbr loop3
            sub ra16, 128, ra16;    -128 back to start point
            sub rb16, 128, rb16;    -128 back to start point
            ret
t12:        ld 0, ra3;              accumulater ra3
            ld rb6, rb1
            push 8;                 accumulate 64 pixels
      loop23:
            push 2;                 accululate 1 line (8 pixels)
      loop22:
            #2 nop
            ld ra16, addra
            ld rb16, addrb
            #3 nop
            sub *addra, *addrb, ra2
            #3 nop
            abs ra2, ra2;
            #3 nop
            ld ra2, sport
            #2 nop
            push 4;                 accumulate 4 pixels
      loop21:
            ld sport, rb2
            #2 nop
            ssr
            add ra3, rb2, ra3;      accumulater ra3
            dbr loop21
            add ra16, 4, ra16;      +4
            add rb16, 4, rb16;      +4
            dbr loop22
            add ra16, 4, ra16;      12-4x2=4
            add rb16, 4, rb16;      12-4x2=4
            dbr loop23
            sub ra16, 96, ra16;     -96 back to start point
            sub rb16, 96, rb16;     -96 back to start point
            ret
t2:
      loopn0:  #3 nsr||ssr||io *mccr%
            push 12;    12
      loopm0:     nsr||ssr||io *mccr%
            ld nport, *mcaw(1)
            ld sport, *mcbw(1)
            dbr loopm0
            ldeamc 49, 1, mc_stride, mc_stride; -3-12+64-1 read,(64x64)
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
            dbr loopn0
            ret
t3:
loopn01:    #3 nsr||ssr||io *mccr%;
```

```
            push 16;     16
      loopm01: nsr||ssr||io *mccr%
            ld nport, *mcaw(1)
            ld sport, *mcbw(1)
            dbr loopm01
            ldeamc 45, 1, mc_stride, mc_stride; -3-16+64-1 read,(64x64)
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
            dbr loopn01
ret
t4:         ld 0, rb6
      loop5:
            push 5;            5colum loop
      loop4:
            call t12
                  ldcr 1110b, acm
                  ifeq ra3, 0, 0
                  #3 nop
                  bpa jump1
                  restore
                  ldcr 0000b, acm
            add rb6, 1, rb6;  next begining point of frameB(address+1)
            add rb16, 1, rb16; next beginning point of frameB(memB's
address+1)
            nop
            dbr loop4
#3 nop
            add rb6, 7, rb6; 12in block address of frameB
            add rb16, 7, rb16;      8 memB+8
            dbr loop5
#3 nop
            ld 9999, rb1;
            ifeq rb1, 9999, 0
            #3 nop
            bpa jump2
jump1:            pop;              loop4
                  pop;              loop5
jump2:            restore
                  ldcr 0000b, acm
            ld rb1, nport
            #3 nop
            io *mcdw%
            add ra1, 1, ra1
ret
t5:         ld 0, rb6
      loop15:
            push 9;            9colum loop
      loop14:
            call t1
                  ldcr 1110b, acm
                  ifeq ra3, 0, 0
```

```
                #3 nop
                bpa jumpl1
                restore
                ldcr 0000b, acm
            add rb6, 1, rb6;  next begining point of frameB(address+1)
            add rb16, 1, rb16; next beginning point of frameB(memB's
address+1)
            nop
            dbr loopl4
#3 nop
            add rb6, 7, rb6; 16 in block address of frameB
            add rb16, 7, rb16; 8 memB+8
            dbr loopl5
#3 nop
            ld 9999, rb1;
            ifeq rb1, 9999, 0
            #3 nop
            bpa jumpl2
jumpl1:           pop;  loop4
                  pop;  loop5
jumpl2:           restore
                  ldcr 0000b, acm
            ld rb1, nport
            #3 nop
            ic *mcdw%
            add ra1, 1, ra1
ret
.end
```

2) GFSM

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;     mocentl.asm                                                        ;
;     mocentl.asm       parallel accumulate, (call)                      ;
;                                                                        ;
; PULSE motion estimation.asm GFSM(8x8block,16x16area,64x64image)        ;
;           frameA is new picture, frameB is old picture                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

            .text
      pull:
            ldcr 2, port1config;
                        (90 2) port 1 as input, synchroneous mode
            ldcr 1, port1incon;
                        (A0 1) port 1 connect to north channel (input)
            ldcr 0, port2config;    (91 0) port 2 as input
            ldcr 2, port2incon;
                        (A1 2) port 2 connect to south channel (input)
            ldcr 1, port3config
            ldcr 1, port3outcon
            ldcr 3, port4config
            ldcr 2, port4outcon

            ldcr 3, bootcontrol;    address purt1 for modulo counter

            ldeamc 0, 2097152, mc_min, mc_min;
            ldeamc  4095, 2101248, mc_max, mc_max;
                        (64x64+start=4096+2097152)
            ldeamc 0, 2097152, mc_start, mc_start;  2**21=2097152
            ldeamc 1, 1, mc_stride, mc_stride;

;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 1, row 1
            ldiamc 0, 0, 143, mcar; 48+96=144
            ldiamc 0, 0, 143, mcaw; 48+96=144
            ldiamc 0, 0, 143, mcbr; 48+96=144
            ldiamc 0, 0, 143, mcbw; 48+96=144
;$$$$$$$$$$ input frame A, B 128-64-16=48 pixels
            push 4;           4x12  4 lines
            call t2
            ld 0, ral;        vector counter number
            ld 0, rbl;        16k, temporal vector address position
;$$$$$$$$$$ input frame A, B 128-32=96 pixels
            push 8;           8+8
            call t2
;!!!!!!!!col 1, row 1,  calculate
            ld 0, ral6;       memA's address
            ld 0, rbl6;       memB's address
            push 5;           loop
            call t4
;!!!!!!!!!first col, first block end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 1, row 1
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 1, row 2-127
```

```
            ldeamc 256, 2097153, mc_start, mc_start;
                                256(64x4), 2097152+1(vector)
            ldiamc 0, 0, 191, mcar; 96+96=192
            ldiamc 0, 0, 191, mcaw; 96+96=192
            ldiamc 0, 0, 191, mcbr; 96+96=192
            ldiamc 0, 0, 191, mcbw; 96+96=192
            ld 48, ra18;        4x12=48 frameA in memA start point
            ld 0, rb18;         frameB in memB start point
;$$$$$$$$$$ input frame A, B 128-32=96 pixels
            push 8;             8x12  8 lines
            call t2
            push 6;             column 1 loops(128-2 for boundry)
      loopo:
;$$$$$$$$$$ input frame A, B 96 pixels
            push 8;             8x12
            call t2
;!!!!!!!!!col 1, 2-127 block calculate
            #2 nop
            ld ra18, ra16;      memA's address
            ld rb18, rb16;      memB's address
            push 9;             9line loop
            call t4
;!!!!!!! One block calculate end
            add ra18, 96, ra18;      8x12
            and ra19, 003fh, ra18
            add rb18, 96, rb18;      8x12
            and rb18, 003fh, rb18
            dbr loopo
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 1, row 2-127
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 1, row 128
;$$$$$$$$$$ input frame A, B 48 pixels
            push 4;
            call t2
;!!!!!!!!!col 1, 128 block calculate
            #2 nop
            ld ra18, ra16;      memA's address
            ld rb18, rb16;      memB's address
            push 5;             9line loop
            call t4
;!!!!!!!!! calculate end
            ldeamc -4092, 1, mc_stride, mc_stride;-4096+(12-4)-3-1=-
4092
;old-4024=-64x(64-1)-8 or +64-8  pointed to next col. Start point
(64x64)
            #3 nop;!
            io *mccr$
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 1, row 128
;end of col. 1 !!!!!!!!!!!!!!!!!!!!!!!!!
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 2-127, row 1
            push 6;     126col.s
loopcol:
```

```
            ldiamc 0, 0, 191, mcar;        64+128=192
            ldiamc 0, 0, 191, mcaw;        64+128=192
            ldiamc 0, 0, 191, mcbr;        64+128=192
            ldiamc 0, 0, 191, mcbw;        64+128=192
;$$$$$$$$$$ input frame A, B 16x4=64 pixels
            push 4;                        4x16  4 lines
            call t3
;$$$$$$$$$$ input frame A, B 128 pixels
            push 8;            8+8
            call t3
;!!!!!!!!!col 2-127, row 1,  calculate
            ld 4, ra16;          (+4) memA's address
            ld 0, rb16;          memB's address
            push 5;              5line loop
            call t5
;!!!!!!!!!calculate end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 2-127, row 1
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 2-127, row 2-127
            ldeamc -512, 1, mc_stride, mc_stride; -510      -64x8-3-1
            #3 nop;!
            io *mccr1
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
            ldiamc 0, 0, 255, mcar;        128+128=256
            ldiamc 0, 0, 255, mcaw;        128+128=256
            ldiamc 0, 0, 255, mcbr;        +128=256
                  ldiamc 0, 0, 255, mcbw; 128+128=256
            ld 68, ra18;         4x16+4=68 frameA in memA start point
            ld 68, rb18; 0       frameB in memB start point
;$$$$$$$$$$ input frame A, B 128 pixels
            push 8;              8x16  8 lines
            call t3
            push 6;              column 1 loops(128-2 for boundry)
      loopo1:
;$$$$$$$$$$ input frame A, B 128 pixels
            push 8;              8x16
            call t3
;!!!!!!!!!col 2-127, 2-127 block calculate
            #2 nop
            ld ra18, ra16;       memA's address
            ld rb18, rb16;       memB's address
            ;;;;;;;;;;;;;
            ;push 9;     9line loop
            call t6;             call t5
            ;;;;;;;;;;;;;
;!!!!!!!!! calculate end
            add ra18, 128, ra18;     8x16
            and ra18, 00ffh, ra18
            add rb18, 128, rb18;     8x16
            and rb18, 00ffh, rb18
            dbr loopo1;
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 2-127, row 2-127
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 2-127, row 128
```

```
;$$$$$$$$$$ input frame A, B 64 pixels
            push 4;           4
            call t3
;!!!!!!!!!col 2-127, 128 block calculate
            #2 nop
            ld ra18, ra16;    memA's address
            ld rb18, rb16;    memB's address
            push 5;           5line loop
            call t5
;!!!!!!!!!! calculate end
            ldeamc -4088, 1, mc_stride, mc_stride; -4096+8-3-1=-4088
;old-4024=-64x(64-1)-8 or +64-8  pointed to next col. Start point
(64x64)
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop;           now is pointed to next line begin
            dbr loopcol
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 2-127, row 128
;end of col. 2-127 !!!!!!!!!!!!!!!!!!!!!!!!
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 128, row 1
            ldiamc 0, 0, 143, mcar;       48+96=144
            ldiamc 0, 0, 143, mcaw;       48+96=144
            ldiamc 0, 0, 143, mcbr;       48+96=144
            ldiamc 0, 0, 143, mcbw;       48+96=144
;$$$$$$$$$$ input frame A, B 12x4=48 pixels
            push 4;           4x12  4 lines
            call t2
;$$$$$$$$$$ input frame A, B 96 pixels
            push 8;           8+8
            call t2
;!!!!!!!!!col 128, row 1,  calculate
            ld 4, ra16;       (+4) memA's address
            ld 0, rb16;       memB's address
            push 5;           5line loop
            call t4
;!!!!!!!!!! calculate end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 128, row 1
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 128, row 2-127
            ldeamc -512, 1, mc_stride, mc_stride; -510      -64x8-3-1
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
            ldiamc 0, 0, 191, mcar;       96+96=192
            ldiamc 0, 0, 191, mcaw;       96+96=192
            ldiamc 0, 0, 191, mcbr;       96+96=192
            ldiamc 0, 0, 191, mcbw;       96+96=192
            ld 52, ra18;      4x12+4=52 frameA in memA start point
            ld 0, rb18;       frameB in memB start point
;$$$$$$$$$$ input frame A, B 128-32=96 pixels
            push 8;           8x12  8 lines
```

```
            call t2
            push 6;             column 1 loops(128-2 for boundry)
      loopo2:
;$$$$$$$$$$ input frame A, B 96 pixels
            push 8;             8x12
            call t2
;!!!!!!!!!col 128, 2-127 block calculate
            #2 nop
            ld ra18, ra16;      memA's address
            ld rb18, rb16;      memB's address
            push 9;             9line loop
            call t4
;!!!!!!!!!! calculate end
            add ra18, 96, ra18;      8x12
            and ra18, 003fh, ra18
            add rb18, 96, rb18;      8x12
            and rb18, 003fh, rb18
            dbr loopo2;
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 128, row 2-127
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@start col 128, row 128
;$$$$$$$$$$ input frame A, B 48 pixels
            push 4;             4
            call t2
;!!!!!!!!!col 1, 128 block calculate
            #2 nop
            ld ra18, ra16;      memA's address
            ld rb18, rb16;      memB's address
            push 5;             5line loop
            call t4
;!!!!!!!!!! calculate end
;@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@end col 128, row 128
;end of col. 128 !!!!!!!!!!!!!!!!!!!!!!!!
end

t1:         ld 0, ra3;  accumulater ra3
            ld rb6, rb1
            push 8;             accumulate 64 pixels
      loop3:
            push 2;             accululate 1 line (8 pixels)
      loop2:
            #2 nop
            ld ra16, addra
            ld rb16, addrb
            #3 nop
            sub *addra, *addrb, rb2;       ra2
            #3 nop
abs rb2, rb2;
#3 nop
add rb2, ra3, ra3
;----
            add ra16, 4, ra16;       +4
            and ra16, 00ffh,ra16
            add rb16, 4, rb16;       +4
            and rb16, 00ffh, rb16
```

```
            dbr loop2
      ;----
            add ra16, 8, ra16;       16-4x2=8
            and ra16, 00ffh,ra16
            add rb16, 8, rb16;       16-4x2=8
            and rb16, 00ffh, rb16
            dbr loop3
      ;----
      ld ra3, sport
push 3
loop1:
      add ra3, sport, ra3
      ssr;                       add before ssr
dbr loop1
            sub ra16, 128, ra16;     -128 back to start point
                  iflt ra16, 0, 0;   (if ra16<0, +256)
                  #3 nop
                  add ra16, 256, ra16
                  else
                  nop
                  restore
            sub rb16, 128, rb16;     -128 back to start point
                  iflt rb16, 0, 0;   (if rb16<0, +256)
                  #3 nop
                  add rb16, 256, rb16
                  else
                  nop
                  restore
            ret
t12:        ld 0, ra3;               accumulater ra3
            ld rb6, rb1
            push 8;                  accumulate 64 pixels
      loop23:
            push 2;                  accululate 1 line (8 pixels)
      loop22:
            #2 nop
            ld ra16, addra
            ld rb16, addrb
            #3 nop
            sub *addra, *addrb, ra2
            #3 nop
      abs rb2, rb2;
      #3 nop
      add rb2, ra3, ra3
            add ra16, 4, ra16;       +4
            and ra16, 003fh, ra16
            add rb16, 4, rb16;       +4
            and rb16, 003fh, rb16
            dbr loop22
            add ra16, 4, ra16;       12-4x2=4
            and ra16, 003fh, ra16
            add rb16, 4, rb16;       12-4x2=4
            and rb16, 003fh, rb16
            dbr loop23
```

```
      ld ra3, sport
push 3
loop21:
      add ra3, sport, ra3
      ssr
dbr loop21
            sub ra16, 96, ra16;      -96 back to start point
                  iflt ra16, 0, 0;   (if ra16<0, +192)
                  #3 nop
                  add ra16, 192, ra16
                  else
                  nop
                  restore
            sub rb16, 96, rb16;      -96 back to start point
                  iflt rb16, 0, 0;   (if rb16<0, +192)
                  #3 nop
                  add rb16, 192, rb16
                  else
                  nop
                  restore
            ret
t2:
      loopn0:  #3 nsr||ssr||io *mccr%
            push 12;     12
      loopm0:     nsr||ssr||io *mccr%
            ld nport, *mcaw(1)
            ld sport, *mcbw(1)
            dbr loopm0
            ldeamc 49, 1, mc_stride, mc_stride; -3-12+64-1 read,(64x64)
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
            dbr loopn0
            ret
t3:
loopn01:    #3 nsr||ssr||io *mccr%;
            push 16;     16
      loopm01: nsr||ssr||io *mccr%
            ld nport, *mcaw(1)
            ld sport, *mcbw(1)
            dbr loopm01
            ldeamc 45, 1, mc_stride, mc_stride; -3-16+64-1 read,(64x64)
            #3 nop;!
            io *mccr%
            nop;!
            ldeamc 1, 1, mc_stride, mc_stride
            #2 nop; now is pointed to next line begin
            dbr loopn01
ret
t4:         ld 0, rb6
      loop5:
            push 5;            5colum loop
```

```
        loop4:
              call t12
                    ldcr 1110b, acm
                    ifeq ra3, 0, 0;;;;;;;;;;;;;;;;;;;;
                    #3 nop
                    bpa jump1
                    restore;;;;;;;;;;;;;;;;;;;;;;;;;;;
                    ldcr 0000b, acm
              add rb6, 1, rb6;  next begining point of frameB(address+1)
              add rb16, 1, rb16;      next beginning point of
                                      frameB(memB's address+1)
              and rb16, 003fh, rb16
              nop
              dbr loop4
              #3 nop
              add rb6, 7, rb6;        12in block address of frameB
              add rb16, 7, rb16;      8 memB+8
              and rb16, 003fh, rb16
              dbr loop5
              #3 nop
              ld 9999, rb1;
              ifeq rb1, 9999, 0
              #3 nop
              bpa jump2
jump1:              pop;              loop4
                    pop;              loop5
jump2:              restore
                    ldcr 0000b, acm
              ld 0, sport
              ld rb1, nport
              #3 nop
              io *mcdw%
              add ra1, 1, ra1
              ret
t5:           ld 0, rb6
        loop15:
              push 9;           9colum loop
        loop14:
              call t1
                    ldcr 1110b, acm
                    ifeq ra3, 0, 0;;;;;;;;;;;;;
                    #3 nop
                    bpa jump11
                    restore;;;;;;;;;;;;;;;;;;;;
                    ldcr 0000b, acm
              add rb6, 1, rb6;  next begining point of frameB(address+1)
              add rb16, 1, rb16; next beginning point of frameB(memB's
                                 address+1)
              and rb16, 00ffh, rb16
              nop
              dbr loop14
              #3 nop
              add rb6, 7, rb6;        16 in block address of frameB
              add rb16, 7, rb16;      8 memB+8
```

```
            and rb16, 00ffh, rb16
            dbr loop15
            #3 nop
            ld 9999, rb1;
            ifeq rb1, 9999, 0
            #3 nop
            bpa jump12
jump11:           pop;          loop14
                  pop;          loop15
jump12:           restore
                  ldcr 0000b, acm
            ld 0, sport
            ld rb1, nport
            #3 nop
            io *mcdw%
            add ra1, 1, ra1
            ret
t6:         ;center
            ld 68, rb6
;                       0 rb16 already in center
            call t1
                  ldcr 1110b, acm
                  ifeq ra3, 0, 0;;;;;;;;;;;;;
                  #3 nop
                  bpa jump22
                  restore;;;;;;;;;;;;;;;;;;;;
                  ldcr 0000b, acm
            add rb16, -17, rb16;    next begining point of memB
                  iflt rb16, 0, 0;  (if rb16<0, +256)
                  #3 nop
                  add rb16, 256, rb16
                  else
                  nop
                  restore
            add rb6, -17, rb6;      next begining point of frameB
;;;3x3
            push 3
      loop211:
            push 3
      loop20:
            call t1
                  ldcr 1110b, acm
                  ifeq ra3, 0, 0;;;;;;;;;;;;;
                  #3 nop
                  bpa jump21
                  restore;;;;;;;;;;;;;;;;;;;;
                  ldcr 0000b, acm
            add rb16, 1, rb16
            and rb16, 00ffh, rb16
            add rb6, 1, rb6
            dbr loop20
            add rb16, 13, rb16;     16-3
            and rb16, 00ffh, rb16
            add rb6, 13, rb6
```

```
            dbr loop211

            add rb16, -65, rb16;-16x4-1=-65 next begining point of memB
                  iflt rb16, 0, 0;  (if rb16<0, +256)
                  #3 nop
                  add rb16, 256, rb16
                  else
                  nop
                  restore
            add rb6, -65, rb6;      next begining point of frameB
;;;5x5
            push 5
      loop30:
            call t1
                  ldcr 1110b, acm
                  ifeq ra3, 0, 0;;;;;;;;;;;;;
                  #3 nop
                  bpa jump23
                  restore;;;;;;;;;;;;;;;;;;;;
                  ldcr 0000b, acm
            add rb16, 1, rb16
            and rb16, 00ffh, rb16
            add rb6, 1, rb6
            dbr loop30

            add rb16, 59, rb16;     16x4-5=59
            and rb16, 00ffh, rb16
            add rb6, 59, rb6
            push 5
      loop31:
            call t1
                  ldcr 1110b, acm
                  ifeq ra3, 0, 0;;;;;;;;;;;;;
                  #3 nop
                  bpa jump23
                  restore;;;;;;;;;;;;;;;;;;;;
                  ldcr 0000b, acm
            add rb16, 1, rb16
            and rb16, 00ffh, rb16
            add rb6, 1, rb6
            dbr loop31
            add rb16,-53, rb16;     -16x3-5=-53
            iflt rb16, 0, 0;  (if rb16<0, +256)
                  #3 nop
                  add rb16, 256, rb16
                  else
                  nop
                  restore
            add rb6, -53, rb6
            push 3
      loop32:
      call t1
                  ldcr 1110b, acm
                  ifeq ra3, 0, 0;;;;;;;;;;;;;
```

```
                #3 nop
                bpa jump23
                restore;;;;;;;;;;;;;;;;;;;
                ldcr 0000b, acm
            add rb16, 4, rb16
            and rb16, 00ffh, rb16
            add rb6, 4, rb6
      call t1
                ldcr 1110b, acm
                ifeq ra3, 0, 0;;;;;;;;;;;;;
                #3 nop
                bpa jump23
                restore;;;;;;;;;;;;;;;;;;;
                ldcr 0000b, acm
            add rb16, 12, rb16;     16-4=12
            and rb16, 00ffh, rb16
            add rb6, 12, rb6
            dbr loop32
            add rb16, -97, rb16;-16x6-1=-97     next begining point of
memB
                iflt rb16, 0, 0;(if rb16<0, +256)
                #3 nop
                add rb16, 256, rb16
                else
                nop
                restore
            add rb6, -97, rb6;      next begining point of frameB
;;;7x7

            push 7
      loop40:
            call t1
                ldcr 1110b, acm
                ifeq ra3, 0, 0;;;;;;;;;;;;;
                #3 nop
                bpa jump23
                restore;;;;;;;;;;;;;;;;;;;
                ldcr 0000b, acm
            add rb16, 1, rb16
            and rb16, 00ffh, rb16
            add rb6, 1, rb6
            dbr loop40

            add rb16, 89, rb16
;16x6-7=89
            and rb16, 00ffh, rb16
            add rb6, 89, rb6
            push 7
      loop41:
            call t1
                ldcr 1110b, acm
                ifeq ra3, 0, 0;;;;;;;;;;;;;
                #3 nop
                bpa jump23
```

```
                restore;;;;;;;;;;;;;;;;;;;
                ldcr 0000b, acm
          add rb16, 1, rb16
          and rb16, 00ffh, rb16
          add rb6, 1, rb6
          dbr loop41
          add rb16, -87, rb16;-16x5-7=-87
          iflt rb16, 0, 0;(if rb16<0, +256)
                #3 nop
                add rb16, 256, rb16
                else
                nop
                restore
          add rb6, -87, rb6
          push 5
    loop42:
    call t1
                ldcr 1110b, acm
                ifeq ra3, 0, 0;;;;;;;;;;;;;
                #3 nop
                bpa jump23
                restore;;;;;;;;;;;;;;;;;;;
                ldcr 0000b, acm
          add rb16, 6, rb16
          and rb16, 00ffh, rb16
          add rb6, 6, rb6
    call t1
                ldcr 1110b, acm
                ifeq ra3, 0, 0;;;;;;;;;;;;;
                #3 nop
                bpa jump23
                restore;;;;;;;;;;;;;;;;;;;
                ldcr 0000b, acm
          add rb16, 10, rb16;      16-6=10
          and rb16, 00ffh, rb16
          add rb6, 10, rb6
          dbr loop42
          add rb16, -113, rb16;    -16x7-1=-113 next begining point of
                                    memB
                iflt rb16, 0, 0;   (if rb16<0, +256)
                #3 nop
                add rb16, 256, rb16
                else
                nop
                restore
          add rb6, -113, rb6;      next begining point of frameB

;;;9x9
          push 9
    loop50:
          call t1
                ldcr 1110b, acm
                ifeq ra3, 0, 0;;;;;;;;;;;;;
                #3 nop
```

```
            bpa jump23
            restore;;;;;;;;;;;;;;;;;;;;
            ldcr 0000b, acm
      add rb16, 1, rb16
      and rb16, 00ffh, rb16
      add rb6, 1, rb6
      dbr loop50

      add rb16, 119, rb16;    16x8-9=119
      and rb16, 00ffh, rb16
      add rb6, 119, rb6
      push 7
loop51:
      call t1
            ldcr 1110b, acm
            ifeq ra3, 0, 0;;;;;;;;;;;;;
            #3 nop
            bpa jump23
            restore;;;;;;;;;;;;;;;;;;;;
            ldcr 0000b, acm
      add rb16, 1, rb16
      and rb16, 00ffh, rb16
      add rb6, 1, rb6
      dbr loop51
      add rb16, -121, rb16;   -16x7-9=-121
      iflt rb16, 0, 0;        (if rb16<0, +256)
            #3 nop
            add rb16, 256, rb16
            else
            nop
            restore
      add rb6, -121, rb6
      push 7
loop52:
call t1
            ldcr 1110b, acm
            ifeq ra3, 0, 0;;;;;;;;;;;;;
            #3 nop
            bpa jump23
            restore;;;;;;;;;;;;;;;;;;;;
            ldcr 0000b, acm
      add rb16, 8, rb16
      and rb16, 00ffh, rb16
      add rb6, 8, rb6
call t1
            ldcr 1110b, acm
            ifeq ra3, 0, 0;;;;;;;;;;;;;
            #3 nop
            bpa jump23
            restore;;;;;;;;;;;;;;;;;;;;
            ldcr 0000b, acm
      add rb16, 8, rb16;      16-8=8
      and rb16, 00ffh, rb16
      add rb6, 8, rb6
```

```
                dbr loop52

                ld 9999, rb1;
                ifeq rb1, 9999, 0
                #3 nop
                bpa jump22
jump21:               pop;
jump23:               pop;
jump22:               restore
                      ldcr 0000b, acm
                ld 0, sport
                ld rb1, nport
                #3 nop
                io *mcdw3
                add ra1, 1, ra1
                ret
.end
```

Appendix G Result of motion estimation

1)

```
qhsim mogood.asm          out.aaa---sun, house, plan & man
      portgoodsun2
16hours                   23,252,180ns              bourassa

# 1162560 instructions ...
# - - - - - - -  Program completed with branchic halt at 23251970 ns
# using 1162565 instructions
# Testbench completed with 0 errors at 23252180 ns
# ** Failure: Simulation Succesfully Completed
#    Time: 23252180 ns  Iteration: 2  Instance:/run
# Break at 2-model_1chip_3mem.vhd line 950
QHSIM 2>

-2147483648
0   0  48  48   0  48  36  48
0  34  32   0   0   0   0   0
2  18 999   1   0  71  71  56
2  34  21   0   0  70  70  66
0   0   0   7  68   4   7  69
0   0   7   7  68  68   2  66
0  68  68   4  67  68   4  68
36 999 52  52  51  52  52  52
-2147483648
-2147483648
```

2)

```
portgood3d1
qhsim       mogood3.asm cut.aaa(sun house plane & man two picture is
different)

# 1545750 instructions ...
# - - - - - - -  Program completed with branchic halt at 30915870 ns
# using 1545760 instructions
# Testbench completed with 0 errors at 30916080 ns
# ** Failure: Simulation Succesfully Completed
#    Time: 30916080 ns  Iteration: 2  Instance:/run
# Break at 2-model_1chip_3mem.vhd line 950
QHSIM 3>
-2147483648
0   0   0   0   0   0   0   0
2  66  66   0   0   0   0   0
2  32 999   5   0  71  71  71
2  66   4   0   0 999 999 999
0   0   7   7  68  68   4  68
6  70  68  35  68  68  68  68
6  70   6  68  68  68   4  68
0 999   0   0  52  52  52  52
-2147483648
-2147483648
-2147483648
```

3)
portgoodshpm
qhsim mogood.asm out.aaa2(sun house plane & man two picture is same)

```
# 1078610 instructions ...
# - - - - - -  Program completed with branchio halt at 21573050 ns
# using 1078619 instructions
# Testbench completed with 0 errors at 21573260 ns
# ** Failure: Simulation Succesfully Completed
#    Time: 21573260 ns  Iteration: 2  Instance:/run
# Break at 2-model_1chip_3mem.vhd line 950
QHSIM 2>

<12 hours

-2147483648
0   0  48  48   0  48  36  48
4  68  16   0   0   0   0   0
4  18  68   1   0   0  68   0
3  68   6   0   0  68  68  68
0   0   0   7  68   4  68  68
0   0   7   7  68  68   2  66
0  68  68   4  67  68   4  68
0  52  52  52  51  52  52  52
-2147483648
-2147483648
```

4)
centers1 Jan. 29, 1998
qhsim mocenter.asm out.aaa2(sun house plane & man two picture is same)

```
# 329910 instructions ...
# - - - - - -  Program completed with branchio halt at 6599030 ns
# using 329918 instructions
# Testbench completed with 0 errors at 6599240 ns
# ** Failure: Simulation Succesfully Completed
#    Time: 6599240 ns  Iteration: 2  Instance:/run
# Break at 2-model_1chip_3mem.vhd line 950
QHSIM 2>
-2147483648
0  48   0   0   0   0   0   0
4  68  68  68  68  68  68   0
4  68  68  68  68  68  68   0
2  68  68  68  68  68  68   0
0  68  68  68  68  68  68   0
0  68  68  68  68  68  68   0
4  68  68  68  68  68  68   0
0  52   0   0  52  52  52  52
-2147483648
-2147483648
-2147483648
```

Appendix H DCT program for PULSE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                         ;
;     dct3.asm    8x8 DCT at 64x64 frame                                 ;
;          parallel DCT (16 times faster than using one PE)               ;
;     F(u,v)=C(u)C(v)/4{sigma x=0->7{sigma y=0->7 f(x,y)                 ;
;           cos[(2x+1)u*3.14/16]cos[2y+1)v3.14/16]}}                      ;
;     condition: c(u), c(v)=1, when u, v > 0; C(u), c(v)=0,707,          ;
;           when u, v = 0.                                                ;
;     table:       cos[(2x+1)u3.14/16]      x:0->7, u:0->7               ;
;          1=128: 0.707=90; 1=256: 0.707=181                              ;
;                                                                         ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          .text
          u .set ra1
          v .set ra2
          y .set rb3
          x .set rb4
          cu .set ra5
          cv .set rb6
          t1 .set r7
          t1h .set rb7
          t2 .set r8;
          t2h .set rb8
          t3 .set r9
          t31 .set ra9
          t4 .set r10
          t4h .set rb10
          u1 .set ra11
          v1 .set ra12
          cu1 .set rb11
          cv1 .set rb12
     pull:
          ldcr 2, port1config;     (90 2) port 1 as input,
                                   synchroneous mode
          ldcr 1, port1incon;      (A0 1) port 1 connect to north
                                    channel (input)
          ldcr 0, port2config;     (91 0) port 2 as input
          ldcr 2, port2incon;      (A1 2) port 2 connect to south
                                    channel (input)
          ldcr 1, port3config
          ldcr 1, port3outcon
          ldcr 3, port4config
          ldcr 2, port4outcon

          ldcr 3, bootcontrol;     address purt1 for modulo counter

          ldeamc 0, 2097152, mc_min, mc_min;
          ldeamc  4095, 2101248, mc_max, mc_max;
     (64x64+start=4096+2097152)
          ldeamc 0, 2097152, mc_start, mc_start;  2**21=2097152
```

```
            ldeamc 1, 1, mc_stride, mc_stride;

            ldiamc 64, 0, 255, mcar;
            ldiamc 0, 0, 255, mcaw;
            ldiamc 0, 0, 255, mcbr;
            ldiamc 0, 0, 255, mcbw;
;*********************************************************
;---------------------------------------------------------
;input table(start from 0 to 63) & input data(start from 64!!!+4)
            call s1;    load table and constant
            ;load constant; u1=0(PE0),u1=8(PE1),u1=16(PE2),u1=24(PE3)
            ;           v1=0(PE0), v1=8(PE1), v1=16(PE2), v1=24(PE3)
            ;           cu1=0.707(PE0)180, cu1=1(PE1~PE3)255
            ;           cv1=0.707(PE0)180, cv1=1(PE1~PE3)255
            ld *mcar(1), u1
            ld *mcar(1), v1
            ld *mcar(1), cu1
            ld *mcar(1), cv1
;-----------------------------------------------------------------
ld cv1, cv
push 8;2;8
ld 0, v;0
loopv:
      ld cu1, cu
      push 2;8
      [illegible]
      loopu:
            mult 0, ra0, acc; 0->acc
            ld 0, y
            push 8
            loopy:
                  ld 0, x
                  push 8
                  loopx:
                        [illegible]; max table value is 65535
                        [illegible]; max table value is 65535
                        #3 nop
                        mult *addra, *addrb, t1; make the value arrive
                                                 high 16 bit in t1.
                        #3 nop
                        macc *mcar(1), t1h;      mcar*t11+acc->acc(high
                                                 16 bit)
                        add x, 1, x
                        #2 nop
                  dbr loopx
                  add y, 1, y
            dbr loopy
            madd 0, ra0, acc, t2;   acc->t2(high 16 bit)
            mult cv, cu, t3;  make the value arrive low 16 bit in t3
            #3 nop
            mult t2h, t31, t4; make the value arrive high 16 bit in t4
            ;-------out
            #2 nop
            ld 0, sport
```

```
            ld t4h, nport
            #3 nop
            io *mcdw%
            #3 nsr||io *mcdw%
            ;-------
      ldiamc 68, 0, 255, mcar;return to start point f(x,y),mean f(0, 0)
      ld 181, cu;             !!!
      add u, 32, u;           4
      dbr loopu
ld 181, cv;                   !!!
add v, 8, v;1
dbr loopv

;*****************************************************
end
s1:   ;load table, constant and data
      push 132;68;64    After table has 4 values(u1, v1, cu1, cv1)
      loop1: #4 nsr||ssr||io *mccr%;
            ld nport, *mcaw(1)
            ld nport, *mcbw(1)
            ;ld sport, *mcbw(1)
      dbr loop1
ret
.end
```

Appendix I IDCT program for PULSE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;      idct.asm    8x8 DCT at 64x64 frame                                       ;
;      parallel DCT (16 times faster than using one PE)                         ;
;      F(u,v)=C(u)C(v)/4{sigma x=0->7{sigma y=0->7 f(x,y)                       ;
;             cos[(2x+1)u*3.14/16]cos[2y+1)v3.14/16]}}                          ;
;      f(x,y)=sigma x=0->7{sigma y=0->7 C(u)C(v)/4*F(u,v)*                      ;
;             cos[(2x+1)u*3.14/16]cos[2y+1)v3.14/16]}                           ;
;            condition: c(u), c(v)=1, when u, v > 0;                            ;
;                       C(u), c(v)=0,707, when u, v = 0.                        ;
;      table:      cos[(2x+1)u3.14/16]     x:0->7, u:0->7                       ;
;                  1=128: 0.707=90; 1=256: 0.707=181                            ;
;                                                                               ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

            .text
            u .set ra1
            v .set ra2
            y .set rb3
            x .set rb4
            cu .set ra5
            cv .set rb6
            t1 .set r7
            t1h .set rb7
            t2 .set r8;
            t2h .set rb8
            t3 .set r9
            t3h .set rb9
            t3l .set ra9
            t4 .set r10
            t4h .set rb10
            u1 .set ra11
            v1 .set ra12
            cu1 .set rb11
            cv1 .set rb12
      pull:
            ldcr 2, port1config;     (90 2) port 1 as input,
                                     synchroneous mode
            ldcr 1, port1incon;      (A0 1) port 1 connect to north
                                     channel (input)
            ldcr 0, port2config;     (91 0) port 2 as input
            ldcr 2, port2incon;      (A1 2) port 2 connect to south
                                     channel (input)

            ldcr 1, port3config
            ldcr 1, port3outcon
            ldcr 3, port4config
            ldcr 2, port4outcon

            ldcr 3, bootcontrol;     address purt1 for modulo counter

            ldeamc 0, 2097152, mc_min, mc_min;
```

```
            ldeamc  4095, 2101248, mc_max, mc_max;
                                    (64x64+start=4096+2097152)
            ldeamc 0, 2097152, mc_start, mc_start;  2**21=2097152
            ldeamc 1, 1, mc_stride, mc_stride;

            ldiamc 64, 0, 255, mcar;
            ldiamc 0, 0, 255, mcaw;
            ldiamc 0, 0, 255, mcbr;
            ldiamc 0, 0, 255, mcbw;
;*****************************************************
;--------------------------
;input table (start from 0 to 63) & input data(start from 64!!!+4)
            call s1;    load table and constant
            ;load constant; u1=0(PE0), u1=8(PE1), u1=16(PE2),
u1=24(PE3)
            ;           v1=0(PE0), v1=8(PE1), v1=16(PE2), v1=24(PE3)
            ;           cu1=0.707(PE0)180, cu1=1(PE1~PE3)255
            ;           cv1=0.707(PE0)180, cv1=1(PE1~PE3)255

            ld *mcar(1), u1
            ld *mcar(1), v1
            ld *mcar(1), cu1
            ld *mcar(1), cv1
;------------------------------
push 8;                         2;8
ld 0, y;                        0
loopy:

      push 2;8
      ld u1,0x;0->x(PE0),1->x(PE1),2->x(PE2),3->x(PE3)
      loopx:
            mult 0, ra0, acc;       0->acc
            ld cv1, cv
            ld 0, v
            push 8
            loopv:
                  ld cu1, cu
                  ld 0, u
                  push 8
                  loopu:
                        add [redacted]; max table value is 65535
                        add [redacted]; max table value is 65535
                        #3 nop
                        mult *addra, *addrb, t1; make the value arrive
                                                 high 16 bit in t1.
                        #3 nop
      ld t1h, rb15;
                        mult *mcar(1), t1h, t4;mcar*t1h->t4(high 16bit)
                        add u, 8, u;
                        #2 nop
mult cv, cu, t3;              make the value arrive low 16 bit in t3
#3 nop
      ld t4h, rb16;
      ld t3l, rb17;
```

```
macc t4h, t3l;                  t4h*t3l+acc->acc(high 16 bit)
                  ld 181, cu; !!!
                  dbr loopu
                  add v, 8, v
                  ld 181, cv; !!!
            dbr loopv
            madd 0, ra0, acc, t2;   acc->t2(high 16 bit)
            ;-------out
            #3 nop
            ld 0, sport
            ld t2h, nport
            #3 nop
            io *mcdw1
            #3 nsr||io *mcdw1
            ;-------
      ldiamc 68, 0, 255, mcar;return to start point f(x,y),mean f(0, 0)
      add x, 4, x;4
      dbr loopx
add y, 1, y;1
dbr loopy
;*****************************************************
end
s1:   ;load table, constant and data
      push 132;68;64    After table has 4 values(u1, v1, cu1, cv1)
      loop1: #4 nsr||ssr||io *mccr1;
            ld nport, *mcaw(1)
            ld nport, *mcbw(1)
      dbr loop1
ret
.end
```

Appendix J cosine table

```
 32767  32767  32767  32767  32767  32767  32767  32767  32767  32767
 32767  32767  32767  32767  32767  32767  32767  32767  32767  32767
 32767  32767  32767  32767  32767  32767  32767  32767  32767  32767
 32767  32767
 32138  32138  32138  32138  27250  27250  27250  27250  18217  18217
 18217  18217   6414   6414   6414   6414  -6363  -6363  -6363  -6363
-18174 -18174 -18174 -18174 -27221 -27221 -27221 -27221 -32127 -32127
-32127 -32127
 30275  30275  30275  30275  12557  12557  12557  12557 -12509 -12509
-12509 -12509 -30255 -30255 -30255 -30255 -30295 -30295 -30295 -30295
-12605 -12605 -12605 -12605  12460  12460  12460  12460  30235  30235
 30235  30235
 27250  27250  27250  27250  -6363  -6363  -6363  -6363 -32127 -32127
-32127 -32127 -18261 -18261 -18261 -18261  18131  18131  18131  18131
 32158  32158  32158  32158   6517   6517   6517   6517 -27162 -27162
-27162 -27162
 23178  23178  23178  23178 -23142 -23142 -23142 -23142 -23215 -23215
-23215 -23215  23105  23105  23105  23105  23252  23252  23252  23252
-23068 -23068 -23068 -23068 -23289 -23289 -23289 -23289  23030  23030
 23030  23030
 18217  18217  18217  18217 -32127 -32127 -32127 -32127   6312   6312
  6312   6312  27308  27308  27308  27308 -27162 -27162 -27162 -27162
 -6568  -6568  -6568  -6568  32178  32178  32178  32178 -18000 -18000
-18000 -18000
 12557  12557  12557  12557 -30295 -30295 -30295 -30295  30235  30235
 30235  30235 -12412 -12412 -12412 -12412 -12701 -12701 -12701 -12701
 30354  30354  30354  30354 -30174 -30174 -30174 -30174  12267  12267
 12267  12267
  6414   6414   6414   6414 -18261 -18261 -18261 -18261  27308  27308
 27308  27308 -32168 -32168 -32168 -32168  32096  32096  32096  32096
-27104 -27104 -27104 -27104  17956  17956  17956  17956  -6056  -6056
 -6056  -6056
```

Appendix K DCT program in C++

```
////////////////////////////////////////////////////////////
//DCT (each pixel = 1 or input from outside)(8x8)     dct3-1.cxx  /
//                                                               /
//g++ -I /usr/local/opt/FSFlibg++/lib/g++-include dct3-1.cxx     /
//F(u,v)=C(u)C(v)/4{sigma x=0->7{sigma y=0->7 f(x,y)             /
//    cos[(2x+1)u*3.14/16]cos[2y+1)v3.14/16]}}                   /
//          condition: c(u), c(v)=1, when u, v > 0;              /
//                     C(u), c(v)=0,707, when u, v = 0.          /
////////////////////////////////////////////////////////////////
# include <stdio.h>
# include <fstream.h>
# include <math.h>
# include <iomanip.h>
# include <stdlib.h>
main()
{
      const float PI = 3.14159, N=8.0;
      int x, y, u, v, i, j, m, n;
      float p, q, r, s, cu, cv, a[8][8] ,b, c[8][8];

/*generate original matrix
      for(m=0; m<8; ++m)
        for(n=0; n<8; ++n)
            a[m][n]=1;*/

/* read a[m][n] from out.b8*/
      ifstream input("out.b8");
      //ifstream input("in.a1");
      char A[5];
      for(j=0; j<8; ++j)
         for(i=0; i<8; ++i)
         {
           input.read((char*)&A, sizeof(A));
           a[i][j] = atoi(A);
           }
/*--------------------------*/
      cv=0.707;
      //cv=0.35;
      for(v=0; v<8; ++v)
      {
        cu=0.707;
        //cv=0.35;
        for(u=0; u<8; ++u)
         {
          b=0.0;
          for(y=0; y<8; ++y)
             for(x=0; x<8; ++x)
            {
              p=(PI*(2*x+1)*u)/16;
              r=cos(p);
              q=(PI*(2*y+1)*v/16);
              s=cos(q);
```

```
                b=a[x][y]*r*s+b;
                //y[0][0]=0;
               // y[i+e][j+f]=x*100000000;
              }
            c[u][v]=(cu*cv*b/4)/3.95;

            cu=1.0;
          }
          cv=1.0;
        }

ofstream out ("out.bl");
for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     out<<setw(5)<<int(a[i][j])<<" ";
     out<< endl;
  }

/*ofstream out ("out.p8"); */
for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     out<<setw(5)<<(c[i][j])<<"  ";
     out<<" #"<< endl;
  }
```

Appendix L IDCT program in C++

1) IDCT program of C++

```
////////////////////////////////////////////////////////////////////
//IDCT              idct3-1.cxx                                     /
//g++ -I /usr/local/opt/FSFlibg++/lib/g++-include idct3-1.cxx       /
//DCT (each pixel = 1)(8x8)                                         /
//f(x,y)={sigma u=0->7{sigma v=0->7 *C(u)C(v)/4*F(u,v)*             /
//        cos[(2x+1)u*3.14/16]*cos[2y+1)v3.14/16]}}                 /
//           condition: c(u), c(v)=1, when u, v > 0;                /
//                      C(u), c(v)=0,707, when u, v = 0.            /
////////////////////////////////////////////////////////////////////
# include <stdio.h>
# include <fstream.h>
# include <math.h>
# include <iomanip.h>
# include <stdlib.h>
main()
{
      const float PI = 3.14159, N=8.0;
      int x, y, u, v, i, j, m, n;
      float p, q, r, s, cu, cv, a[8][8] ,b, c[8][8];

/*generate original matrix
      for(m=0; m<8; ++m)
        for(n=0; n<8; ++n)
            a[m][n]=1;*/

/* read a[m][n] from out.b9*/
      ifstream input("out.b1");
      //char A[8];
      //ifstream input("out.c1");
      char A[8];
      for(j=0; j<8; ++j)
         for(i=0; i<8; ++i)
         {
           input.read((char*)&A, sizeof(A));
           a[i][j] = atof(A);
           }
/*--------------------------*/
      for(y=0; y<8; ++y)
      {
        for(x=0; x<8; ++x)
         {
          b=0.0;
          cv=0.707;
          for(v=0; v<8; ++v)
             {
             cu=0.707;
             for(u=0; u<8; ++u)
            {
              p=((2*x+1)*u*3.1416)/16;
              r=cos(p);
```

```
                q=((2*y+1)*v*3.1416/16);
                s=cos(q);
                b=(cu*cv/4)*a[u][v]*r*s+b;
                //y[0][0]=0;
               // y[i+e][j+f]=x*100000000;
              cu=1.0;
              }
               cv=1.0;
               }
            c[x][y]=b*3.95;
          }
        }

ofstream out ("out.dl");
for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     //out<<setw(5)<<int(a[i][j])<<" ";
     out<<setw(8)<<a[i][j]<<" ";
     out<< endl;
  }

/*ofstream out ("out.p10"); */
for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     out<<setw(5)<<(c[i][j])<<"  ";
     out<<" #"<< endl;
  }
}
```

2)C++ IDCT program simulate PULSE assemble IDCT

```
////////////////////////////////////////////////////////////////////
//IDCT              idct3-1.cxx                                     /
//g++ -I /usr/local/opt/FSFlibg++/lib/g++-include idct3-1asm.cxx    /
//DCT (each pixel = 1)(8x8)                                         /
//f(x,y)=(sigma u=0->7(sigma v=0->7 *C(u)C(v)/4*F(u,v)*             /
//    cos[(2x+1)u*3.14/16]*cos[2y+1)v3.14/16]})                     /
//          condition: c(u), c(v)=1, when u, v > 0;                 /
//                     C(u), c(v)=0,707, when u, v = 0.             /
////////////////////////////////////////////////////////////////////
# include <stdio.h>
# include <fstream.h>
# include <math.h>
# include <iomanip.h>
# include <stdlib.h>
main()

{
      const float PI = 3.14159, N=8.0;
      int x, y, u, v, i, j, m, n;
      float p, q, r, s, cu, cv, z, cosine[9][9], a[8][16] ,b, c[8][8];

/* read cos table
      ifstream input("out.b1");
      char A[8];
      for(j=0; j<8; ++j)
         for(i=0; i<8; ++i)
         {
           input.read((char*)&A, sizeof(A));
           cos[i][j] = atof(A);
           }*/

/* creat cos table*/
/*    for (i=0; i<8; ++i)
         for(j=0; j<8; ++j)
                  {z=(2*j+1)*i*3.14/16;
                  cosine[j][i]=cos(z)*32767;
           }
*/
/* read a[m][n] from out.b9*/
      ifstream input("idctinput");
      char A[8];
      for(j=0; j<8; ++j)
         for(i=0; i<8; ++i)
         {
           input.read((char*)&A, sizeof(A));
           a[i][j] = atof(A);
           }

      for(j=8; j<16; ++j)
         for(i=0; i<8; ++i)
         {
           input.read((char*)&A, sizeof(A));
```

```
            a[i][j] = atof(A);
            }

       for(j=0; j<8; ++j)
          for(i=0; i<8; ++i)
          { cosine[i][j]=a[i][j+8];
          }
/*--------------------------*/
       for(y=0; y<8; ++y)
       {
         for(x=0; x<8; ++x)
          {
           b=0.0;
           //cv=0.707;
           cv=140;
           for(v=0; v<8; ++v)
              {
              //cu=0.707;
              cu=140;
              for(u=0; u<8; ++u)
             {
              // p=((2*x+1)*u*3.1416)/16;
              // r=cos(p);
              r=cosine[x][u];
              // q=((2*y+1)*v*3.1416/16);
              //s=cos(q);
              s=cosine[y][v];
               //b=(cu*cv/4)*a[u][v]*r*s+b;
               b=(cu*cv*a[u][v]*r*s)/4294967296.0+b;
               //y[0][0]=0;
              // y[i+e][j+f]=x*100000000;
             cu=180;
             }
              cv=180;
              }
            //c[x][y]=b*3.95*0.93/1000000000;
            c[x][y]=b*3.95/100000;
          }
        }

/*ofstream out ("out.dl");
for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     //out<<setw(5)<<int(a[i][j])<<" ";
     out<<setw(8)<<cosine[i][j]<<" ";
     out<< endl;
   }

for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     //out<<setw(5)<<int(a[i][j])<<" ";
     out<<setw(8)<<a[i][j]<<" ";
```

```
      out<< endl;
   }
*/

ofstream out ("out.dl");
for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     //out<<setw(5)<<int(a[i][j])<<" ";
     out<<setw(8)<<a[i][j]<<" ";
     out<< endl;
   }
   for(j=8; j<16; ++j)
  {
   for(i=0; i<8; ++i)
     //out<<setw(5)<<int(a[i][j])<<" ";
     out<<setw(8)<<a[i][j]<<" ";
     out<< endl;
   }

for(j=0; j<8; ++j)
  {
   for(i=0; i<8; ++i)
     out<<setw(5)<<(c[i][j])<<"  ";
     out<<" #"<< endl;
  }
}
```

Appendix M cosine table generate program in C++

```
////////////////////////////////////////////////////////////
/*great-costb.cxx 64 table element + 4 constant*/           /
/                                                          /
////////////////////////////////////////////////////////////
# include <stdio.h>
# include <fstream.h>
# include <iomanip.h>
# include <iostream.h>
# include <math.h>

main()
{
      int i,j;
      float A[9][9], z;
      ofstream out ("cut.b1");

      for (i=0; i<8; ++i)
         for(j=0; j<8; ++j)
                  {z=(2*j+1)*i*3.14/16;
                  A[i][j]=cos(z)*32767;
                  }
      for (i=0; i<8; ++i)
         {
          for(j=0; j<8; ++j)
          {
/* 4 time output to file for PULSE 4 PE same*/
          out<<setw(4)<<int(A[i][j]);
          out<< endl;
          out<<setw(4)<<int(A[i][j]);
          out<< endl;
          out<<setw(4)<<int(A[i][j]);
          out<< endl;
          out<<setw(4)<<int(A[i][j]);
          out<< endl;
         /* cut<<"\n"<< endl;*/
          }
      /*    cut<<setiosflags(ios::right)
             <<setw(4)<<A[i][j]<<" ";
          out<<setiosflags(ios::right)
             <<setw(4)<<A[i][8];
             out<<endl;*/
          }
}
```

Appendix N Data transfer programs for C40 and PULSE

1) Read/write data from/to global memory with C40 (C language):----------

```
#include "tms320c40.cmd"

#define global process1

#define local   process2

#define TRUE 1

#define FALSE 0


int value;

main()

{

}


global()

        {

        int mem_mal;

        /* Initialize Part */

        trace_on();

        set_trace_level (GLOBAL,2);

        idle (5);

        load_reg (GLOBAL_CTRL, 0x37843fa0);
```

```
        /* Read/write data from/to global memory */
        mem_val = read (0x81080001);
        load_pin (STAT, 0x1);
        edle (20);
        write (0x8108000b, 0x0f0f0f0f);
        load_pin (STAT, 0x5);
        idle (20);
        mem_val = read (0x0007000r); /* Dummy read */
        idle (10);
        synch();
}
```

2) Read/write data from/to local memory with C40 (C language):----------

```
#include "tms320c40.cmd"
#define global process1
#define local   process2
#define TRUE 1
#define FALSE 0

int value;
main()
{
```

```
}

local()
{
        trace_on();
        set_trace_level (LOCAL,2);
        set_trace_level (PERIPH,2);
        idle (20);
        /* Read/write data from/to data memory*/
        value = read (0x0500001);
        load_pin (LSTAT, 0x1);
        idle (20);
        write (0x040000b, 0x0f0f0f0f);
        load_pin (LSTAT, 0x5);
        idle (20);
        value = read (0x0007000F); /* Dummy read */
        idle (10);
        synch();
}
```

3) Read/write data from/to local memory with PULSE (PULSE assembly language):------

```
.text
```

```
ldcr 3, bootcontrol; Use internal program memory and Mcc for address_1

;Read local memory
ldcr 0, port2config; port2: input synchronous mode
ldcr 0, port3config; port3: input synchronous mode
ldcr 2, port2incon; port2 => South Channel
ldcr 1, port3incon; port3 => North Channel

ldeamc 0, 50000bh, mc_start, mc_start;
ldeamc 0, 0, mc_min, mc_min
ldeamc 4096, 50FFFFh, mc_max, mc_max
ldeamc 1, 1, mc_stride, mc_stride

io *mccr%
ld nport, ra1
ld sport, rb1
#30 nop

;Write local memory
ldcr 1, port2config; port2: output synchronous mode
ldcr 1, port3config; port3: output synchronous mode
ldcr 1, port3outcon; port3 => North Channel
```

```
ldeamc 0, 40000ah, mc_start, mc_start;
ldeamc 0, 0, mc_min, mc_min
ldeamc 4096, 40ffffh, mc_max, mc_mas
ldeamc 1, 1, mc_stride, mc_stride

ld 65535, nport
ld 255, sport
#4 nop
io *mcdw%
#32 mop
```

4) Read/write data from/to program memory with C40 (C language):----------

```
#include "tms320c40.cmd"
#define global process1
#define local  process2
#define TRUE 1
#define FALSE 0
int value;
main()
{
}
```

```
global()
        {
        int mem_mal;
        /* Initialize Part */
        trace_on();
        set_trace_level (GLOBAL,2);
        idle (5);
        load_reg (GLOBAL_CTRL, 0x37843fa0);

        /* Read data from program memory */
        idle (25);
        mem_val = read (0x00600001); /* Read at address 1 of band0 */
        load_pin (STAT, 0x9);
        idle (5);
        mem_val = read (0x00700001); /* Read at address 1 of bank1 *;
        load _pin (STAT, 0x9);
        idle (5);
        mem)_val = read (0x00800001); /* Read at address 1 of bank2 */
        load_pin (STAT, 0x9);
        /* Write data to program memory*/
        idle (5);
```

```
write (0x00300001, 0xffffffff); /* Write at address 1 of bank0 */
load_pin (STAT, 0xd);
idle (5);
write (0x00400001, 0xf0f0f0f0); /* Write at address 1 of bank1 */
load_pin (STAT, 0xd);
idle (5);
write (0x00500001, 0x0f0f0f0f); /* Write at address 1 of bank2 */
load_pin (STAT, 0xd);
synch();
}
```

Appendix O PULSE vs Competitors

Features	CNAPS	SHARC	TI C80	Oxford A236	PULSE v1
Architecture	SIMD	Single-processor	MIMD	SIMD	SIMD
Clock Frequency	20~25 MHz	33~40MHz	50MHz	40MHz	54MHz
Number of Processor on the chip	64 PNs. Without Controller	Single floating point Processor	One floating-point Processor	One Controller 4 fixed-point processors	One Controller 4 fixed-point processors
Inter-PE Communication support	Very Weak, 5 Mbytes/s	N/A	Cross Bar memory access	No inter-PE communication	Strong, Multichannel 432 Mbytes/s
Parallel Operations in the PE	Very weak Multiply-acc	Strong Two adders one multiplier	Strong 3-input ALU Multiply-acc	Weak Multiply-acc multiply-add	Very strong 3-input, 3-output ALU, Mult-add-acc add-acc med-add
Non-linear processing	No	Weak Only max, min and clip of two data	Weak	No	Very strong Max, Min, Med Rank-order Index ranking Core function chip
Application Mapping	Very restricted	.....	Very flexible, But very hard to program	Very restricted	Flexible and easy to program
Scalability	Scalable	Scalable	No	Scalable	Scalable
External Memory and I/O Interface	No memory interface, 8-bit I/O	4 buses, 64-bit datapath to SRAM 10 DMA	Single 64-bit Bus shared by all the processors for	400 Mbytes/s 32-bit sync. Memory 2 40 Mbytes/s	Two buses 432 Mb/s sync. Memory 4 108 Mb/s

		Channels. 160 Mbytes/s	data and program	DMA ports	data ports
On-chip memory	4kbytes on each Pn	2Mbits or 4Mbits	50Kbytes	1kbytes Ins. 1kbytes data	2kbytes Ins. 2.5kbytes data
Micro-Instruction	64-bit total 32-bit control 32-bit PN	48-bit	64-bit for parallel proc. 32-bit master	32-bit	64-bit parallel
Instruction-set	Very limited	Rich. extended for non-linear processing	Rich. extended for logic processing	Very limited	Very rich extended for both linear and non-linear proc.
Software Tools	Assembler C compiler Debugger	Assembler C compiler Simulator Debugger Evaluation board	Assembler C compiler Simulator Debugger Evaluation board	Software development kit	Assembler C compiler Simulator Debugger Evaluation board Application libraries
Packaging	200-pin PN 240-pin CSC PGA	240-pin PQFP	305-pin ceramic PGA	208-pin PQFP	240-pin PQFP
Availability	Yes	Yes	Yes	Q2 1996	Q4 1996
Cost	High	High	High	Low	Low